EPIC-1

Workshop on Explicitly Parallel Instruction Computing (EPIC) Architectures and Compiler Techniques

Austin, Texas

December 2nd, 2001

Workshop Proceedings

EPIC-1

Held in Conjunction with MICRO-34
Austin, Texas
December 2, 2001

*http://systems.cs.colorado.edu/*EPIC1/

GENERAL CHAIR

Dan Connors, University of Colorado
University of Colorado

PROGRAM CHAIRS

Carole Dulong, Intel Corporation
Rick Hank, Hewlett-Packard Corporation

PROGRAM COMMITTEE

Santosh Abraham, Sun Microsystems
David August, Princeton University
Brad Calder, University of California-San Diego
Dan Connors, University of Colorado
Tom Conte, North Carolina State University
Jim Dehnert, Transmeta Corporation
Kemal Ebcioglu, IBM Research
Wen-mei Hwu, University of Illinois
Suneel Jain, Hewlett-Packard Corporation
Teresa Johnson, Hewlett-Packard Corporation
Vinod Kathail, Hewlett-Packard Research Labs
Dan Lavery, Intel Corporation
Scott Mahlke, University of Michigan
Krishna Palem, Georgia Tech University
Jim Pierce, Intel Corporation
Nancy Warter, Cal State- Los Angeles

EPIC-1

1st Workshop on EPIC (Explicitly Parallel Instruction Computing) Architectures and Compiler Technology

Final Program

December 2nd, 2001

Sunday, December 2nd

9:00AM	**Welcome-Dan Connors (University of Colorado)-General Chair** **Carole Dulong (Intel Corporation) and** **Rick Hank (Hewlett Packard), Program Co-Chairs** **(Marriott Ballroom)**
9:00-9:30AM	**Keynote Speech: Dr. Wei Li, Principal Engineer and manager of compiler development for the Itanium Processor Family in the Intel Compiler Lab** **"Compiling for the Itanium Architecture: Triumphs, Lessons, and Challenges"**
9:30- 10:00AM	**Session A:**
9:30-10:00AM	**Software pipelining of loops with early exits for the Itanium architecture**. Kalyan Muthujumar (*Intel Technology-India*), Dong-Yuan Chen, Youfeng Wu (*Intel Microprocessor Research Labs*) and Daniel M. Lavery (*Intel Compiler Lab*)
10:00-10:30AM	**Predicate-Based Transformations to Eliminate Control and Data-Irrelevant Cache Misses** Ian Bratt, Alex Settle, and Dan Connors (*University of Colorado-Boulder*)
10:30-11:00AM	**Break**
11:00- 12:30PM	**Session B:**
11:00- 11:30AM	**EPIC Instruction Scheduling Based on Optimal Approaches.** Steve Haga and Rajeev Barua (*ECE-University of Maryland, College Park*)
11:30- 12:00AM	**A Spill Code Reduction Technique for EPIC Architectures.** Virgil Palanciuc, Dragos Badea, and Costel Ilas (*Motorola DSP Center Romania*) and Eric Flamand (*Motorola Metrowerks France*)
12:00-12:30PM	**Incorporating Predicate Information Into Branch Predictors.** Beth Simon, Brad Calder, and Jeanne Ferrante (*University of California- San Diego*)
12:30-2:00PM	**Lunch (provided)**
2:00- 3:30PM	**Session C:**
2:00- 2:30PM	**Fracture Mechanics on the Intel (R) Itanium Architecture (A Case Study).** Gerd Herber and Andrew Dolgert (*Cornell Theory Center*), Maxim Alt, Karen A. Mazurkiewicz, and Lynd Stringer (*Intel Corporation*).
2:30- 3:00PM	**Dynamic Binary Instrumentation on IA-64.** Vinodha Ramasamy and Robert Hundt (*Dynamic Instrumentation Group, Hewlett Packard Company*)
3:00- 3:30PM	**TRITANIUM: Augmenting the Trimaran Compiler Infrastructure To Support IA-64 Code Generation.** Yogesh Chobe, Bhagi Narahari, Rahul Simha (*George Washington University*) and Weng-Fai Wong (*National University of Singapore*)

Software Pipelining of Loops with Early Exits for the Itanium™ Architecture

Kalyan Muthukumar[↑] Dong-Yuan Chen[ξ] Youfeng Wu[ξ] Daniel M. Lavery[√]

[↑]Intel Technology India Pvt Ltd
17 Mahatma Gandhi Road
Bangalore 560 001, India

[ξ]Intel Microprocessor Research Labs
2200 Mission College Blvd
Santa Clara, CA 95052, USA

[√]Intel Compiler Labs
2200 Mission College Blvd
Santa Clara, CA 95052, USA

{kalyan.muthukumar, dong-yuan.chen, youfeng.wu, daniel.m.lavery}@intel.com

ABSTRACT

The Itanium architecture contains many features to enhance parallel execution, such as an explicitly parallel (EPIC) instruction set, large register files, predication, and support for speculation. It also contains features such as register rotation to support efficient software pipelining of loops. Software-pipelining techniques have been shown to significantly improve the performance of loop-intensive scientific programs that have only one exit per loop. However, loops with multiple exits, which are often encountered in control-intensive non-numeric programs, pose a special challenge to such techniques. This paper describes the schema for generating software-pipelined loops using Itanium architecture features. It then presents two new methods for transforming a loop with multiple exits so that it can be efficiently software-pipelined on Itanium. They make control-flow transformations that convert a loop with multiple exits into a loop with a single exit. This is followed by if-conversion to create a loop consisting of a single basic block. These methods are better than existing techniques in that they result in better values of II (Initiation Interval) for pipelined loops and smaller code sizes. One of these methods has been implemented in the Intel optimizing compiler for the Itanium architecture. This method is compared with the technique suggested by Tirumalai *et al* and we show that it performs better on loops of the benchmark programs in SpecInt2000.

1. INTRODUCTION

The Itanium architecture contains many features to enhance parallel execution, such as an explicitly parallel (EPIC) instruction set, large register files, predication [18], and support for speculation [19]. It also contains features such as register rotation [7] to support efficient software pipelining without increasing the code size of the loop. Software pipelining [2][1][3][11][21][12][9] tries to improve the performance of a loop by overlapping the execution of several iterations. This improves the utilization of available hardware resources by increasing the instruction-level parallelism (ILP).

Typically, the loop body that is to be software-pipelined is assumed to consist of a single basic block. This simplifies the task of software pipelining. A loop with control flow in it (and hence containing multiple basic blocks) can be converted to a single basic block by the process of *if-conversion* [8], which removes branches by predicating instructions based on the condition of the branch. However, loops that have early exits (i.e. that have more than one exit branch out of the loop) cannot be converted into a single basic block by this technique alone. Such loops often occur in control-intensive programs (for example a **break** statement in C/C++) and are also created by compiler transformations such as tail duplication that exclude some paths from the pipelined loop [5].

This paper introduces the Itanium™ architecture support for software pipelining and shows how it can be used to efficiently pipeline loops in control-intensive programs, including while loops and loops with early exits. Two new methods for transforming loops with multiple exits to make them pipelineable are also presented. They make control-flow transformations that convert a loop with multiple exits into a loop with a single exit. This is followed by if-conversion to collapse the remaining control flow and create a loop consisting of a single basic block.

The first method is an improvement of the technique by Tirumalai *et al* [15]. It uses a general-purpose register (as in the Tirumalai method) to "remember" which early exit was taken, if any. This register is examined after the loop terminates to determine which exit was taken and where execution should proceed. Our first method takes advantage of parallel compare instructions in the Itanium architecture to reduce the recurrence MII of the pipelined loop. Our second method uses predicate registers to "remember" which early exit was taken, if any. After the loop terminates, the predicate registers are examined and control branches to the target of the appropriate early exit, if necessary. This method reduces both the resource MII and recurrence MII compared to the method proposed by Tirumalai. We have implemented the second method and the technique of Tirumalai in the Intel optimizing compiler for Itanium. We present results of applying these two techniques on the SPEC CINT2000 suite of benchmarks. The method proposed in this paper results in an average 31% improvement in achieved II compared to Tirumalai's method for the early exit loops in SPEC CINT2000.

The rest of this paper is organized as follows. Section 1.1 provides some background and motivates the need to perform transformations on loops with early exits so that they can be efficiently pipelined. Section 2 describes the Itanium architecture features and Section 3 describes the software-pipelining schema using these features. The existing method of Tirumalai is presented in Section 4. Sections 5 and 6 present the

two methods that improve the performance of software-pipelined loops that have early exits and compares them with Tirumalai's method. Section 7 presents the experimental results of applying these techniques to the SPEC CINT2000 benchmarks. Section 8 provides a summary and directions for future work.

1.1 Background and Motivation for Early Exit Transformations

The number of cycles between the start of successive iterations in a software-pipelined loop is called the Initiation Interval (II). Software-pipelined loops have three execution phases: the prolog phase, in which the software pipeline is filled, the steady-state kernel phase, in which the pipeline is full, and the epilog phase, in which the pipeline is drained. The software pipeline is coded as a loop that is very different from the original source code loop. To avoid confusion when discussing loops and loop iterations, we use the term source loop and source iterations to refer back to the original source code loop, and the term kernel loop and kernel iterations to refer to the loop that implements the software pipeline.

In this paper we discuss early exit transformations in the context of modulo scheduling [17], though the transformations may apply to other software pipelining algorithms as well. In the modulo-scheduling algorithm a minimum candidate II is computed prior to scheduling. This candidate II is the maximum of the resource-constrained minimum II (resource MII) and the recurrence-constrained minimum II (recurrence MII). The resource MII is based on the resource usage of the loop and the processor resources available. The recurrence MII is based on the cycles in the dependence graph for the loop and the latencies of the processor.

Loops that have control flow pose a special challenge for modulo scheduling. This is because the modulo-scheduling algorithm has generally been developed to schedule either a single basic block or a trace consisting of a single path through the loop body. Loops that contain control flow within the loop body have been handled by one of three methods: if-conversion [7][14], hierarchical reduction [12], or combining modulo scheduling with other software-pipelining algorithms [13][20]. In addition, infrequent or hazardous paths can be excluded from modulo-scheduling altogether [5]. In Intel's Itanium™ compiler, control flow within the loop body is removed using if-conversion, and tail-duplication is used to exclude some paths from scheduling.

If-conversion alone cannot remove early exits from the loop. Figure 1 shows the overlapped execution of several iterations of a pipelined loop that has one early exit.

The figure shows an early exit branch taken in iteration (i+2). However, some instructions in earlier iterations i and (i+1) have not yet been executed. These are shown by the black areas in the figure. Also instructions from iteration (i+2) that were moved from above to below the early exit during scheduling have not yet been executed. These are shown by the grey area in the figure. Using the pipelining method in [5], an explicit epilog can be created that contains the instructions from the black and gray areas. The target of the early exit is changed to this epilog block. Control then branches from this new block to the original target of the early exit. One such explicit epilog is required for each early exit in the loop. The addition of the epilog blocks increases the code size. Some of the epilogs may be combined, but the largest epilog, that corresponding to the first early exit, always remains. Lavery's approach works well for architectures that do not have predication. However, the predication and software pipelining support in the Itanium architecture make it desirable to investigate methods that remove branches and generate kernel-only code (code with no explicit prologs or epilogs [16]).

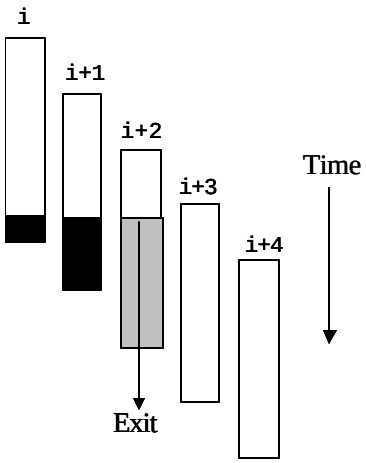


Figure 1. The overlapped execution of a software-pipelined loop with one early exit.

One such method, proposed by Tirumalai et al [15], is to remove the early exit branches but "remember" that the condition for an early exit became true during the execution of the loop. When an early exit occurs, the epilog phase of the pipelined loop is executed in the kernel only code with the appropriate parts of the last source iteration disabled using predication. Upon exit from the loop, control branches to the target of the "remembered" early exit. Thus, the loop with multiple exits is converted to a loop with a single exit. There is still control flow within the loop, but this is removed by the process of if-conversion. Unfortunately, as shown later in this paper, the original transformation method proposed by Tirumalai increases the number of instructions in the loop, increasing the resource MII. The second method proposed in this paper transforms the loop in such a way that it almost always reduces the number of instructions in the original loop. Our method always achieves a resource MII that is equal to or better than the Tirumalai method. It also always achieves a resource MII that is equal to or better than Lavery's method for architectures like Itanium that have separate compare and branch instructions.

2. ITANIUM™ ARCHITECTURE FEATURES

Itanium provides many features to aid the compiler in enhancing and exploiting ILP. These include an explicitly-parallel (EPIC) instruction set, large register files, predication, speculation, and support for software pipelining. An Itanium program must explicitly indicate groups of instructions, called *instruction groups*, that have no register flow or output dependences. Instruction groups are delimited by *architectural stops* in the code. Because instruction groups have no register flow or output dependences, they can be issued in parallel without hardware checks for register dependences between instructions. In the

examples in this paper, architectural stops are indicated by double semicolons (;;). Parallel execution generates many simultaneously live values. The Itanium™ architecture provides 128 general and 128 floating-point registers, so that these results can be kept in registers rather than spilled to memory.

Predication refers to the conditional execution of an instruction based on a boolean source operand called the qualifying predicate. If the qualifying predicate is one, the instruction is executed. If the qualifying predicate is zero, the instruction generally behaves like a no-op. Predicates are assigned values by compare instructions. Predication removes branches and the mispredict penalties associated with them and exposes more ILP [18]. Almost all Itanium instructions have a qualifying predicate. Even conditional branches have qualifying predicates. The branch is taken if the qualifying predicate is one, and not taken otherwise. Itanium provides 64 predicate registers.

Compare instructions in Itanium each have two destination predicate registers. Generally, the first destination is set to one if the compare condition evaluates to true and zero if the condition evaluates to false. The second destination is set to the complement of the first. Three types of compares are used in this paper: conditional, unconditional, and parallel. Conditional compares behave similarly to other predicated Itanium instructions. If the qualifying predicate is zero, the conditional compare leaves both destination predicate registers unchanged. Unconditional compares set both destination predicate registers to zero when the qualifying predicate is zero.

Parallel compares allow the results of two compares to be combined in a way that is analogous to a wired-AND or wired-OR in logic design. Two parallel compares can target the same destination predicate register but still be placed in the same instruction group and issued in parallel. To use the AND-type parallel compare, the destination predicate register is first initialized to one. Then if the condition for a subsequent AND-type compare evaluates to false, the predicate register is set to zero. If the condition evaluates to true, the predicate register is left unchanged. The OR-type parallel compares are similar. The destination predicate register is initialized to zero. Then if the condition for a subsequent OR-type compare evaluates to true, the predicate register is set to one. Parallel compares are useful for implementing the || and && logical operators in C and for computing the conditions for the execution of blocks in nested if-then-else constructs. In the following code, the store is executed only if all three conditions are true:

```
        cmp.unc  p1,p0 = (r1==r2) ;;  // unconditional compare
(p1)    cmp.unc  p2,p0 = (r3==5) ;;   // unconditional compare
(p2)    cmp.unc  p3,p0 = (r6>0) ;;    // unconditional compare
(p3)    st [r4] = r5
```

Predicate register p0 is hardwired to one and when used as a destination, the result is discarded. Using AND-type parallel compares, all three conditions can be evaluated in parallel:

```
        p1 = 1;;
        cmp.and  p1,p0 = (r1==r2)    // AND-type compare
        cmp.and  p1,p0 = (r3==5)     // AND-type compare
        cmp.and  p1,p0 = (r6>0) ;;   // AND-type compare
(p1)    st [r4] = r5
```

Itanium features both control and data speculation. In this paper, we restrict the discussion to control speculation. Control speculation refers to the execution of an instruction before the branch that guards it. Itanium contains features similar to those first proposed in [19] that allow the compiler to perform speculative code motion. Speculative versions of loads defer exceptions by writing a special token into the destination register to indicate the existence of a deferred exception. Most computation instructions propagate deferred exception tokens from the source to the destination registers. Speculation check instructions are used to test for the presence of a deferred exception token. If a token is found, the check instruction branches to recovery code to re-execute the speculative instruction sequence and handle the exception.

3. SOFTWARE PIPELINING IN THE ITANIUM™ ARCHITECTURE

Itanium provides extensive support for software-pipelined loops, including register rotation, and special loop branches and registers. Such features were first seen in the Cydrome Cydra-5 [4][7][6][10]. Register rotation provides a renaming mechanism that eliminates the need for loop unrolling for the purpose of software renaming of registers. Special software-pipelined loop branches support register rotation and, combined with predication, reduce the need to generate separate blocks of code for the prolog and epilog phases. This section describes each of the Itanium™ architecture software pipelining features and shows how they can be used to efficiently pipeline loops in control-intensive programs.

Register rotation renames registers by adding the register number to the value of a rotating register base (rrb) register modulo the size of the rotating register file. The rrb register is decremented when a software-pipelined loop branch is executed at the end of each kernel iteration. Decrementing the rrb register makes the value in register X appear to move to register X+1. If X is the highest numbered rotating register, it's value wraps to the lowest numbered rotating register.

General registers r32-r127, floating-point registers f32-f127, and predicate registers p16-p63 can rotate. Below is an example of register rotation. The swp_branch pseudo-instruction represents a software-pipelined loop branch:

```
L1:     ld4      r35 = [r4],4        // post increment by 4
        st4      [r5] = r37,4        // post increment by 4
        swp_branch  L1 ;;
```

The value that the load writes to r35 is read by the store two iterations (and two rotations) later as r37. In the meantime, two more instances of the load are executed. Because of register rotation, those instances write their results to different registers and do not destroy the value needed by the store.

The rotation of predicate registers serves two purposes. The first, similar to the rotating general and floating-point registers, is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of the pipeline. To do this, a predicate is assigned to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called a *stage predicate.* For counted loops, p16 is architecturally defined to be the predicate for the first stage, p17 is defined to be the predicate for the second stage, etc. A possible pipeline schedule for a source iteration of an example counted loop is shown below (before allocation of general and floating-point registers – capital V in the register name indicates that it is a virtual register) with the

stage predicate assignments. Each stage is one cycle long (II = 1) and a load latency of 2 is assumed:

```
stage 1:  (p16)  ld4   Vr4 = [Vr5],4
stage 2:  (p17)                          // empty stage
stage 3:  (p18)  add   Vr7 = Vr4,r9
stage 4:  (p19)  st4   [Vr6] = Vr7,4
```

A register rotation takes place at the end of each stage (when the software-pipelined loop branch is executed in the kernel loop). Thus a one written to p16 enables the first stage and then is rotated to p17 at the end of the first stage to enable the second stage for the same source iteration. Each one written to p16 sequentially enables all the stages for a new source iteration. This behavior is used to enable or disable the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases.

Generally speaking, each time a software-pipelined loop branch is executed, the following actions take place:

1. A decision is made on whether or not to continue kernel loop execution.
2. The registers are rotated (rrb registers are decremented).
3. p16 is set to a value to control execution of the stages of the software pipeline.
4. Special Loop Count (LC) and Epilog Count (EC) registers are selectively decremented.

There are two types of software-pipelined loop branches: counted (br.ctop) and while (br.wtop). For counted loops, during the prolog and kernel phase, a decision to continue kernel loop execution means that a new source iteration is started. p16 is set to one to enable the stages of the new source iteration. LC is decremented to update the count of remaining source iterations. EC is not modified.

Once LC reaches zero, the epilog phase is entered. During this phase, a decision to continue kernel loop execution means that the software pipeline has not yet been fully drained. p16 is now set to zero because there are no more new source iterations and the instructions that correspond to non-existent source iterations must be disabled. EC contains the count of the remaining execution stages for the last source iteration and is decremented during the epilog. When the EC register reaches one, the pipeline has been fully drained, and the kernel loop is exited.

A pipelined version of the example counted loop, using Itanium software pipelining features, is shown below assuming two memory units and 200 iterations:

```
        mov     lc = 199              // LC = loop count - 1
        mov     ec = 4                // EC = epilog stages + 1
        mov     pr.rot = 1 << 16 ;;   // PR16 = 1, rest = 0

L1: (p16) ld4   r32 = [r5],4
    (p18) add   r35 = r34,r9
    (p19) st4   [r6] = r36,4
          br.ctop L1 ;;
```

There are several differences in the operation of the while loop branch compared to the counted loop branch. The br.wtop does not access LC. Like ordinary conditional branches in the Itanium™ architecture, a qualifying predicate determines the behavior of this branch instead. If the qualifying predicate is one, the br.wtop is taken and another kernel iteration is executed. If the qualifying predicate is zero, then the br.wtop is taken if EC is greater than one (i.e. the pipeline is not yet fully drained). Also, p16 is always set to zero. The reasons for these differences are related to the nature of while loops vs. counted loops.

In a purely counted loop, the br.ctop instruction does not depend on any other instructions in the loop, allowing a decision to be quickly made on whether or not to start a new source iteration. In while loops, the decision to execute another source iteration depends on a more general computation ultimately terminating in a compare that sets the qualifying predicate for the br.wtop branch. The computation may be complex and contain loads or other long latency instructions. Thus the br.wtop branch may not be ready for execution until late in the source iteration, resulting in limited overlap of the iterations.

To increase overlap, subsequent iterations can be started speculatively [15] using Itanium's support for control speculation. The software pipeline stages prior to the one containing the compare that sets the br.wtop's qualifying predicate are called the *speculative stages* of the pipeline because it is not possible for the compare and br.wtop branch to control the execution of these stages. Therefore these stages are not assigned stage predicates. The result of the compare is also used as a stage predicate to control the first non-speculative stage of the pipeline, the stage containing the compare itself. The compare rather than the br.wtop generates the predicate that enables the non-speculative portion of the next source iteration.

A possible software pipeline schedule for one iteration of a simple while loop with II equal to one is shown below (including stage predicate assignments):

```
stage 1:  L1:    ld4   Vr4 = [Vr5],4
stage 2:                             // empty stage
stage 3:  (p18) add   Vr7=Vr4+Vr9 // first non-speculative stage
          (p18) cmp   Vp1,p0 = (Vr4!=0)
stage 4:  (p19) st4   [Vr6] = Vr7,4
          (Vp1) br    L1
```

Note that instructions that already have qualifying predicates (such as the br L1 above) are not assigned stage predicates. The pipelined version of the while loop with II equal to one is shown below. A check for the speculative load is included:

```
          mov     ec = 3      //  # epilog stages + # speculative
                                  pipeline stages
          mov     pr.rot = 1 << 16 ;;  // PR16 = 1, rest = 0
L1:       ld4.s   r32 = [r5],4         // speculative load
(p18)     chk.s   r34, Recovery        // check r34 (result of
                                       ld.s) for exception token
(p18)     add     r35 = r34 + r9
(p18)     cmp     p17,p0 = (r34!=0)
(p19)     st4     [r6] = r36,4
(p17)     br.wtop L1 ;;
```

There is no stage predicate assigned to the load because it is speculative. The compare sets p17. This is the branch predicate for the current source iteration and, after rotation, the stage predicate for the first non-speculative stage (stage three) of the next source iteration. During the prolog, the compare cannot produce its first valid result until the third iteration of the kernel loop. The initialization of the predicates provides a pipeline of predicate values that keeps the compare disabled until the first source iteration reaches stage three. At that point the compare

is enabled and starts generating stage predicates to control the non-speculative stages of the pipeline for the next iteration. Notice that the compare is conditional. If it were unconditional, it would always write a zero to p17 and the pipeline would not get started correctly.

During the last source iteration, the compare result is zero. Therefore, during the next kernel iteration the stage predicate for the compare is zero. The conditional compare stops writing results and the stream of zeros written by the br.wtop branch is rotated in to drain the pipeline.

4. TIRUMALAI METHOD FOR EARLY EXIT TRANSFORMATION

This section describes the method of Tirumalai et al [15]. We describe their method in detail so that it can be compared with our methods on the same example program. We use the loop in Figure 2 as a running example. This control flow graph shows a loop with two blocks A and B. Blocks A and B are *exit blocks* of the loop. The successor block of an exit block that is outside the loop is called the *post-exit* block. In this example, block D is the post-exit block of block A and block C is the post-exit block of block B. For simplicity, we only consider a loop that is in a bottom-test form. The *bottom block* (block B) of the loop must be an exit block. An exit block that is not the bottom block is an *early exit*. We consider innermost loops with only one backedge. If the loop has more than one backedge, the loop can be transformed into a nested loop and the innermost loop will have only one backedge.

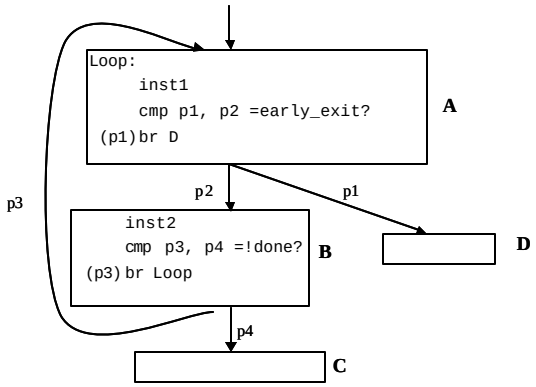


Figure 2. An example loop

Tirumalai et al [15] describes a method that converts a loop with multiple exits into a loop with a single exit. In their method, each exit of the loop is associated with a unique number. When an exit condition is satisfied, its associated number is assigned to a variable, and if the exit is not at the bottom of the loop, control is immediately transferred to the loop exit at the bottom of the loop. After the loop completes and exits from the bottom block, control transfers to the appropriate post-exit block depending on the value in the variable. Figure 3 shows the loop after the control flow transformations of their method.

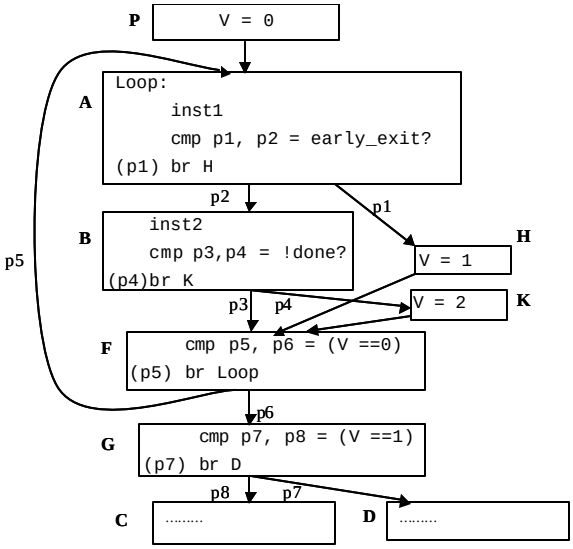


Figure 3. Tirumalai transformation of early exits.

The transformation consists of the following steps:

1. Initialize a new virtual register V to zero in the loop pre-header P.
2. Create a new bottom block F after the original bottom block B.
3. Create a new post-exit block G as the new fall-through block for block F.
4. For the exit from block A, create a new block H. The target of block A is changed from D to the new block H. A new instruction V = 1 is inserted into block H. Block H branches to new bottom block F. Similarly for the exit from block B that branches to the post-exit block C, we create a new block K. The target of block B is changed from C to the new block K. A new instruction V = 2 is inserted into block K. Block K branches to the new bottom block F. In general, for a loop with N exits, N blocks containing assignments to V are added.
5. Block F checks whether or not V is zero. If V is zero, it branches back to loop entry. Otherwise, control falls through to post-exit block G.
6. Block G branches to the appropriate target of the original loop exit depending on the value of V. If (V == 1), then the early exit condition is true and control flows to block D. Otherwise, control falls through to block C. In general, if there are N exits, N-1 branches will be inserted in block G.

Notice that this method adds the following three instructions to the loop: V = 1, V = 2, and cmp p5, p6 = (V == 0). In addition, V is initialized to zero in the loop preheader and the loop post-tail contains the compare of V with one. In general, for a loop with N exits, their method adds (N+1) instructions to the loop, one instruction to the loop preheader and (N-1) compare instructions following the loop exit. These additional instructions often increase the resource MII of the software-pipelined loop and may potentially increase the size of the pipelined kernels.

After the transformation, the loop has a single exit. The loop is then if-converted. Figure 4 shows the result of if-converting the transformed loop.

```
       V = 0
Loop:
       inst1
       cmp p1, p2 = early_exit?
  (p1) V = 1
  (p2) inst2
  (p2) cmp.unc p3,p4 = done?
  (p3) V = 2
       cmp p5, p6 = (V == 0)
  (p5) br Loop
Block G:
       cmp p7, p8 = (V == 1)
  (p7) br D
```

Figure 4. If-converted code for the example in Figure 2.

Notice that the recurrence MII of the loop transformed by this method is at least 4 cycles, assuming that all the instructions in the recurrence cycle have unit latencies. The dependence cycle is shown in Figure 5. The loop-independent edges are caused by flow dependencies, while the loop-carried edge between "cmp p5, p6=" and "cmp p1, p2=" is caused by control dependence. In general, for a loop with N early exits, this method will have a recurrence MII of at least (N+3) cycles.

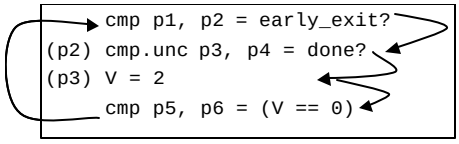


Figure 5. Dependence cycle of the if-converted code in Figure 4.

5. SIMPLE IMPROVEMENTS TO TIRUMALAI METHOD

The Tirumalai algorithm can be improved slightly by removing the assignment "V=2" as shown in Figure 6. This reduces resource usage in the loop and also reduces the recurrence MII by one. There are now two dependence cycles with 3 cycles each for the loop.

```
              V = 0
Loop:
              p5 = 0
              inst1
              cmp p1, p2 = early_exit?
       (p1)   V = 1
       (p2)   inst2
       (p2)   cmp.unc p3,p4 = !done?
       (p3)   cmp p5, p6 = (V == 0)
       (p5)   br Loop
Block_G:
              cmp p7, p8 = (V == 1)
       (p7)   br D
```

Figure 6. Improved code for the example.

We can further improve the above code using the Itanium™ architecture's *parallel compare* instructions. Using AND-type parallel compares, the two compare instructions "(p2) cmp.unc p3,p4 = !done?" and "(p3) cmp p5, p6 = (V == 0)" in Figure 6 can be transformed so that they can be executed in a single cycle. This converts the example loop into the code as shown in Figure 7. Although the transformed loop still has a recurrence MII of 3 cycles, it has only one dependence cycle with 3 cycles. Figure 8 shows the longest dependence cycle. In general, for a loop with N early exits, these improvements will lead to a recurrence MII of at least (N+2) cycles. This is an improvement of 1 cycle over the Tirumalai algorithm.

```
              V = 0
Loop:
              p5 = 1
              inst1
              cmp p1, p2 = early_exit?
       (p1)   V = 1
       (p2)   inst2
       (p2)   cmp.and p5 = !done?
              cmp.and p5 = (V == 0)
       (p5)   br Loop
Block G:
              cmp p7, p8 = (V == 1)
       (p7)   br D
```

Figure 7. Improved early exit transformation with parallel compare.

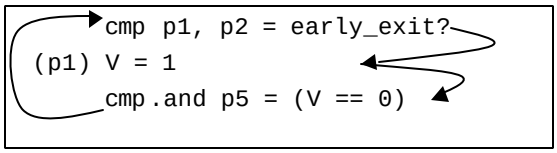


Figure 8. Dependence cycle of the if-converted code in Figure 7.

6. EARLY EXIT TRANSFORMATION USING PREDICATE REGISTERS

This section presents a new method for transforming loops with multiple exits into a single exit loop for efficient software pipelining. In Tirumalai's method, a variable is dedicated to record which early exit in the loop has been taken. A closer examination of the original method reveals that the qualifying predicate register for each early exit branch also carries the taken-exit information. With this insight, we have derived a more efficient method for transforming a multiple-exit loop into a single-exit loop by utilizing the predicate registers and unconditional compare instructions in the Itanium™ architecture. The new method assigns one predicate register to each early exit in the loop. The predicate register is set to one whenever its corresponding early exit is taken. Right after the software-pipelined loop, control flow is redirected to the correct target block depending on which of these predicate registers is one. Since the compare instructions for checking the exit conditions already exist in the loop body, new instructions are only introduced for initializing these predicate registers, which can be eliminated by predicate initialization optimization for most cases. Furthermore, as a result of if-conversion, all the early exit branches are removed, resulting in a decrease in the number of instructions in the loop.

6.1 Transformations Using Predicate Registers

We will use the example loop of Figure 2 to illustrate the method using predicate registers. The transformation consists of two phases. First, the loop is transformed into a single-exit loop. Then the transformed loop body is collapsed into a single basic block using if conversion. Figure 9 shows the loop in Figure 2 after the first phase of the transformation. The first phase of the transformation consists of the following steps.

1. Split the original bottom block B into two blocks -- B and E. The new bottom block E now contains the loop branch while the other instructions remain in Block B.
2. Redirect the targets of all early exits to the new bottom block E.
3. Create a new block F between the new bottom block E and original post-exit block C, one for each early exit. The new block F contains a new branch to the original target of the early exit, that is, block D. The new branch instruction is predicated by the predicate register assigned to that early exit. For example, in Figure 9 block F contains a branch to block D if p1 is one.
4. Initialize the predicate register (p3 in our example) that controls the loop back branch to zero at the beginning of the loop entry block so that control will exit the loop if an early exit is taken.
5. Initialize all predicates for early exit branches to zero in the beginning of the loop entry block.

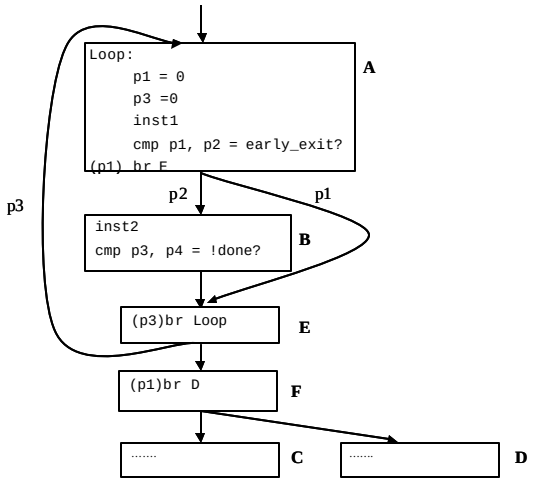


Figure 9. The loop in Figure 2 after transformation using predicate registers.

Note that not all of the predicate initializations introduced in steps 4 and 5 are really necessary. We will discuss how to optimize away these predicate initializations in Section 6.2.

We then apply if conversion to the resulting control flow graph after the first phase of the transformation, collapsing the loop body into a single basic block. Figure 10 shows the loop from Figure 9 after a straightforward if-conversion.

```
Loop:
            p1 = 0
            p3 = 0
            inst1
            cmp p1, p2 = early_exit?
      (p2)  inst2
      (p2)  cmp p3, p4 = !done?
      (p3)  br Loop
Block_F:
       (p1)   br D
```

Figure 10. The if-converted code of Figure 9.

6.2 Optimization of Predicate Initializations

As mentioned before, not all of the predicate initializations are really necessary. Here we describe several approaches to optimize away these predicate initializations. The first optimization is to apply traditional dead code elimination on the if-converted code. For example, in Figure 10, initialization of p1 is killed by the instruction "cmp p1,p2=" before reaching any of its use. Hence "p1=0" is dead code and can be safely removed.

The second optimization makes use of the unconditional form of the compare in the Itanium™ architecture to combine predicate initialization and compare instruction into one single instruction, eliminating the standalone predicate initialization instruction. In the if-converted code of Figure 10, we could replace the compare "(p2) cmp p3,p4=" with an unconditional compare "(p2) cmp.unc p3,p4=", eliminating the initialization of p3 to 0. When the path ABE in Figure 9 is taken, p2 is one and thus p3 will get its value from the result of the compare instruction in block B. When the path AE is taken, p2 is zero and the unconditional compare will reset p3 to zero.

```
Loop:
         inst1
         cmp p1, p2 = early_exit?
   (p2)  inst2
   (p2)  cmp.unc p3, p4 = !done?
   (p3)  br Loop
Block_F:
       (p1)   br D
```

Figure 11. The loop of Figure 10 after predicate initialization optimization

Figure 11 shows the loop body from Figure 10 after applying optimization on predicate initializations. Note that the merging of a predicate initialization "p=0" and a compare instruction "cmp p,q=" into an unconditional compare instruction is applicable if and only if the following condition holds. That is, all uses of the destination predicate p (or q) after the compare instruction "cmp p,q=" in the if-converted code must be either defined by the predicate initialization instruction "p=0" (or "q=0") or defined by the compare instruction "cmp p,q=" in the control flow graph before if conversion. This condition prevents the unconditional compare transformation from killing any other reaching definition to the use of predicate p (or q).

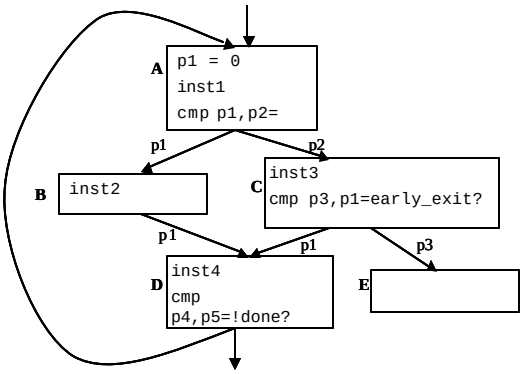
Figure 12 illustrates the necessity of this condition. Figure 12(a) is the control flow graph before if conversion and Figure 12(b) shows the if-converted code with "cmp p3,p1=" in block C turned into an unconditional compare "(p2) cmp.unc p3,p1=" for illustration purposes. Assume that the compare "cmp p1,p2=" results in p1=1 and p2=0, that is, execution is expected to follow the path ABD and the loop back branch is to be taken. When

execution reaches "(p2) cmp.unc p3,p1=", the unconditional compare would reset both p1 and p3 to zero because its qualifying predicate p2 is zero. This would result in an incorrect value of p1=0 when execution reaches "(p1) cmp.unc p4,p5=" and thus the loop would exit prematurely.

The third optimization applies to early exits caused by **break** statements, such as the code segment shown below.

```
while (……) {
      if (cond)
            break;
}
```

In such cases, it is not necessary to "remember" which early exit was taken, because the targets of the early exit branches are the same as the fall-through block of the loop's bottom block. In such cases, the predicates for the early exit branches are not live out of the loop and they need not be computed. Therefore, the predicate initializations for the early exit predicates can be removed and the compares that define such predicates need not be converted into unconditional compares. These cases account for many of the early exits in structured programs, and are well suited for our method.



(a)

```
Loop:
      inst1
      cmp p1, p2 =
  (p1) inst2
  (p2) inst3
  (p2) cmp.unc p3, p1 = early_exit?
  (p1) inst4
  (p1) cmp.unc p4, p5 = !done?
  (p4) br Loop
Post_exit:
  (p3) br E
```

(b)

Figure 12. An example where the predicate initialization cannot be optimized.

6.3 Performance Improvement for Transformation Using Predicate Registers

We evaluate the performance improvement for our new transformation using two metrics commonly used to measure the performance of software pipelined loops, namely the Resource MII and Recurrence MII.

Transformation using predicate registers does not add any new instructions to the loop body except predicate initializations, which could be optimized away in most of the cases. Furthermore, early loop exit branches are moved out of the loop body to the loop post-exit, reducing the number of instructions in the loop body. Hence the transformed loop always has a smaller number of instructions in the loop body in comparison with Tirumalai's algorithm, resulting in the same or a smaller Resource MII.

To illustrate potential improvement in recurrence MII, consider the code in Figure 11. Assume that we are generating kernel-only code for the software-pipelined loop and the latency of compares is one cycle. The two compare instructions in this loop create a dependence cycle. There is a loop-independent data dependence edge from "`cmp p1,p2=early_exit?`" instruction to "`(p2) cmp.unc p3,p4=!done?`" instruction because of the flow dependence on p2. Furthermore a loop-carried control dependence edge from "`(p2) cmp.unc p3,p4=!done?`" instruction to "`cmp p1,p2=early_exit?`" instruction of the next iteration is required if we are generating kernel-only code without explicit epilogs [16]. See Figure 13. The Recurrence MII is at least 2 cycles, which is better than the recurrence MII lower bound in Tirumalai's algorithm, which is at least 4 cycles. In general, for loop with N early exits, our method results in a minimum recurrence MII of (N+1) cycles as compared to the (N+3) cycles required by Tirumalai's algorithm.

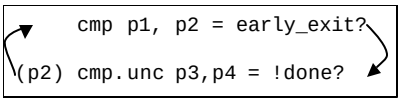


Figure 13. Dependence cycle of the code in Figure 11.

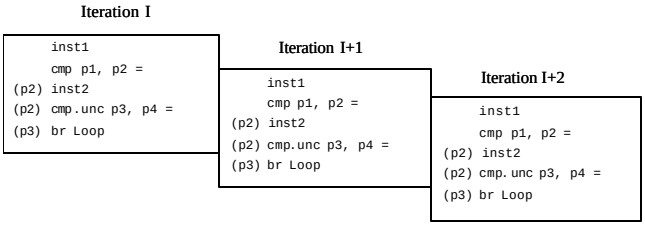


Figure 14. An example illustrating the necessity of the loop-carried control dependence edge.

The necessity of the loop-carried control dependence edge for kernel only code can be illustrated using the example in Figure 14. Here we intentionally break the loop-carried control dependence edge between two consecutive iterations by speculatively executing "cmp p1,p2=" from the $(I+1)^{th}$ iteration before we know that this iteration will be executed. Figure 14 shows the overlapped execution of three iterations of the transformed loop. Assume that, in the I^{th} iteration, predicate p3 is set to zero by "(p2) cmp.unc p3,p4=" because an early exit has been signaled during this iteration which sets the qualifying

predicate p2 to zero. Hence, after iteration I, the software pipelined loop kernel is expected to enter the epilog phase and drain the pipeline by maintaining a zero value in p3. However, the compare instruction "cmp p1,p2=" from the $(I+1)^{th}$ iteration is executed *speculatively* before the unconditional compare "(p2) cmp.unc p3,p4=" in iteration I. The execution of "cmp p1,p2=" from iteration (I+1) may speculatively set p2 to one which could then in turn set p3 to one during the epilog phase. This results in incorrect program behavior because the execution of the software pipeline epilog is not maintained until the pipeline is drained.

If the loop-carried control dependence edge is properly honored, the compare instruction "cmp p1,p2=" in iteration (I+1) would have been qualified by a stage predicate that is set to one only if the execution of iteration (I+1) is really required. Thus predicate p1 and p2 in iteration (I+1) will be properly set to zero during the epilog phase, draining the pipeline properly.

Note that the loop-carried control dependence edge is only necessary for generating kernel-only code. Alternatively, we could generate a single explicit epilog as the schema for the software-pipelined loop, eliminating the need for the loop-carried control dependence edge. In this case, we could break the recurrence cycle in both Tirumalai's and our methods but our method still has a better resource MII. Furthermore, code size may increase if we generate an explicit epilog.

```
ConvertMultipleExitsToSingleExitLoop() {
1.  Split the bottom block into two blocks such that the new bottom
    block NB contains only the loop branch;
2.  Initialize the loop branch predicate to zero in the loop entry block;
3.  FOR every early exit branch B guarded by predicate P and with a
    target block T in the loop DO
    {
4.      Change the target of branch B to the
        new bottom block NB;
5.      IF (B is not caused by a "break") {
6.          Initialize P to zero at the
                beginning of the loop entry block;
7.          Add a new block before the original
                post-exit block containing the
                branch instruction "(P) br T"
    }   }

    /* Now do if conversion and optimization. */
8.  Apply If-conversion to collapse the loop body into a single basic
    block;
9.  Apply dead code elimination to remove dead predicate
    initialization;
10. FOR each compare defining predicates p and q DO {
11.     IF (p is used later and an unconditional
            definition of p at this location kills
            a previous definition of p other than
            "p=0")
12.         CONTINUE;
13.     IF (q is used later and an unconditional
            definition of q at this location kills
            a previous definition of q other than
            "q=0")
14.         CONTINUE;
15.     Convert the compare into an unconditional compare;
16.     Remove the previous predicate initialization "p=0" and
        "q=0", if they exist;

17. }   }
```

Figure 15. Outline of transformation algorithm using predicate registers.

6.4 Algorithm

Figure 15 outlines the algorithm for transforming a loop with multiple exits into a single exit loop utilizing predicate registers in the Itanium™ architecture.

7. EXPERIMENTAL RESULTS

We have implemented both our method (described in Section 6) and the method of Tirumalai *et al* in the Intel optimizing compiler for Itanium. In this section, we compare the results of using these two techniques on the SPEC CINT2000 suite of benchmark programs. **Table 1** shows the percentage improvements obtained by our method over the technique of Tirumalai *et al*. These results were obtained by running the benchmarks with O2 level optimization. Profile feedback was not used.

The second column shows the total number of loops in each benchmark that is pipelined. This includes loops with early exits as well as loops that only have one exit. The third column in the table shows the number of loops with early exits that are pipelined. The fourth column shows the average percentage improvements in the II of the pipelined loops that have early exits. Performance gains in II vary from 18% for **181.mcf** to 44% for **252.eon**. The fifth column presents the gains in execution times on an Itanium 800 MHz machine. On an average, our method provides a 31% improvement in II for the early exit loops and an overall 1% gain in measured performance on an Itanium 800 MHz machine for all of SPEC CINT2000. These results confirm our claim in Section 6 that our method should lead to better loop performance as compared to the technique of Tirumalai *et al*.

SPEC Cint2000 Benchmarks	Number of pipelined loops	Number of pipelined loops with early exits	Percentage improvement in II for early exit loops	Percentage improvement in execution time on Itanium
164.gzip	50	7	25%	0.6%
175.vpr	90	20	43%	0.0%
176.gcc	707	402	28%	0.9%
181.mcf	22	5	18%	0.0%
186.crafty	84	20	37%	2.2%
197.parser	72	22	28%	0.2%
252.eon	95	15	44%	-0.8%
253.perlbmk	201	108	33%	1.4%
254.gap	578	244	36%	1.4%
255.vortex	39	21	41%	2.1%
256.bzip2	43	5	21%	2.7%
300.vortex	339	70	20%	0.5%
Total/Average	2320	939	31%	1.0%

Table 1 . Experimental Results

The percentage improvements in II for the early exit loops are computed *statically* i.e. they do not take into account the number of times these loops are executed. All loops are given the same weight. The values of II for both the methods are those achieved when compiling for the Itanium™ processor. An interesting point that we noticed is that our method provides a greater percentage improvement in II for smaller values of II. This can be explained as follows. First, a loop with larger II is probably constrained by other recurrence cycles or by a resource MII that is a greater than the recurrence cycle caused by the compares of the early exit branches. Therefore, the improvement due to our method does not have an effect on the scheduled II. This was observed in the case of the **300.twolf** benchmark. Second, the reduction in recurrence MII due to our method has a larger impact on loops with smaller II. A reduction of II from 4 cycles to 2 cycles produces a performance gain of 50% while reducing the II from 12 to 10 cycles leads to a gain of only 17%. The

255.vortex benchmark has the lowest average II of all the benchmarks and has one of the greatest performance gains of 41% due to our method.

8. CONCLUSIONS

We have presented two new methods for transforming a loop with multiple exits so that it can be efficiently software-pipelined on an EPIC architecture like the Itanium™ architecture. They make control flow transformations followed by if-conversion and some optimizations to reduce the number of instructions in the loop. They take advantage of Itanium features such as predicate registers, parallel compares, unconditional compares, etc. They are better than existing techniques for pipelining loops with multiple exits because: (1) the resource MII and recurrence MII of pipelined loops with these new methods are smaller, and (2) they do not generate explicit epilogs, which means reduced code size. The experimental results that we have obtained with one of our methods shows significant improvements in the achieved II values of pipelined loops in SPEC CINT2000 benchmarks over the technique of Tirumalai et al. These methods enable software-pipelining of control-intensive non-numeric programs such as SPEC CINT2000 and database engines and improve their performance on EPIC architectures such as Itanium.

9. ACKNOWLEDGEMENTS

Comments from anonymous reviewers helped improve the presentation of the paper. We deeply appreciate the support and encouragement of this work from the management and our colleagues at the Intel Compiler Labs and the Intel Microprocessor Research Labs.

10. REFERENCES

[1] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In Proceedings of PLDI'88 (June 1988), 308-317.

[2] A. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. IEEE Computer (Sept. 1981).

[3] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture (Oct. 1981), 183-198.

[4] B. R. Rau and D. W. L. Yen and W. Yen and R. A. Towle. The Cydra 5 Departmental Supercomputer". IEEE Computer. Vol 22, No. 1. January 1989.

[5] D.M. Lavery and W.W. Hwu. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In Proceedings of the MICRO-29 (Dec. 1996), 126-137.

[6] G. R. Beck, David W.L. Yen, and Thomas L. Anderson. The Cydra 5 Minisupercomputer: Architecture and Implementation. The Journal of Supercomputing. Volume 7, No. 1/2, 1993. Pages 143-180.

[7] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped Loop Support in the Cydra 5. In Proceedings of ASPLOS'89 (Apr. 1989), 26-38.

[8] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In Proceedings of POPL'83 (Jan. 1983), 177-189.

[9] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In Proceedings of PLDI'96 (May 1996), 1-11.

[10] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing. Volume 7, No. 1/2, 1993. Pages 181-228.

[11] K. Ebcioglu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture (Dec. 1987), 69-79.

[12] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In Proceedings of PLDI'88 (June 1988), 318-328.

[13] M.G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In Proceedings of MICRO-29 (Dec. 1996), 262-273.

[14] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In Proceedings of MICRO-25 (Dec. 1992), 170-179.

[15] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of Loops with Exits on Pipelined Architectures. In Proceedings of Supercomputing '90 (Dec. 1990), 200-212.

[16] Ramakrishna Rau, Michael S Schlansker, and P. P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. MICRO-25 (1992).

[17] Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In Proceedings of MICRO-27 (1994).

[18] S. A. Mahlke, R. E. Hank, J.E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In Proceedings of ISCA'95 (June 1995), 138-150.

[19] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel Scheduling for Superscalar and VLIW Processors. In Proceedings of ASPLOS'92 (Oct. 1992), 238-247.

[20] S. Shim and S-K Moon. Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows. In Proceedings of MICRO-31 (Nov. 1998), 93-102.

[21] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Sofeware Pipelining. ACM Computing Surveys, 27(3) (Sept. 1995).

Predicate-Based Transformations to Eliminate Control and Data-Irrelevant Cache Misses

Ian R. Bratt, Alex Settle, and Daniel A. Connors
Department of Electrical and Computer Engineering
University of Colorado
Boulder, Colorado 80309

ABSTRACT

The performance of modern processors is increasingly dependent on their ability to execute multiple instructions per cycle. Explicitly Parallel Instruction Computing (EPIC) architectures can achieve high performance by using the compiler to express program instruction level parallelism (ILP) directly to the hardware. The predicated execution feature is critical to the success of the EPIC architecture approach because it allows the compiler to explicitly overlap the execution of independent control paths. An advantage of predicated execution is the elimination of hard-to-predict branches by executing both paths of a branch in a single code sequence. However, there are a number of advantages to predicated execution support other than simply reducing branch mispredictions and enabling the parallel execution of multiple control flow paths. To date, very little research has been performed to investigate radical changes to a program's original data flow and control flow. This paper outlines methods of using predicated execution to substantially improve the efficiency of EPIC memory accesses. Value locality, the repetition of computation values serve as a key attribute to reduce unnecessary memory operations of a program.

1. INTRODUCTION

Memory access penalties have become a major issue for the microprocessor industry to deliver higher performance microprocessors. The growing disparity between processor and memory performance will make cache misses increasingly expensive. Additionally, data caches are not always used efficiently, resulting in large numbers of data cache misses. In numeric programs there are several known compiler techniques for optimizing data cache performance [13]. However, integer (non-numeric) programs often have irregular access patterns that are more difficult for the compiler to optimize.

As memory latencies increase, the importance of cache performance improvements at each level of the memory hierarchy will continue to grow. Rather than strictly trying to improve memory system efficiency with adaptive cache management techniques [9] it makes sense to direct advances in compiler technology to determine whether memory accesses are absolutely necessary to determine the results of program computations. By eliminating unnecessary memory accesses, it is possible to improve memory access efficiency and deal with long memory latencies, utilizing predicated execution to eliminate unnecessary memory requests. Specifically, the goal is to increase data cache effectiveness for integer programs.

Generally, optimizing compilers are successful at eliminating inefficiencies and redundancies within programs by performing iterations of analysis and optimization. The elimination of redundancies in programs at compile time can dramatically improve a program's execution time. Traditional compiler techniques such as constant propagation, common subexpression elimination, loop invariant code removal, conditional branch elimination, and partial redundancy elimination [5] eliminate program redundancy and and improve the efficiency of a program. However, compiler optimization techniques rely on the detection of static redundancy, which requires the complete assertion that the computations be definitely redundant for all executions. However, compiler techniques have limited mechanisms for exploiting dynamic values in programs. Several empirical program behavior studies indicate that many instruction traces are dynamically executed with the same inputs, a form of redundancy known as *value locality* [6] Overall, in order to exploit value locality for the purpose of improving the efficiency of the memory system, hardware support must be available to a processor to selectively disable memory requests that are determined as unnecessary. The EPIC architecture predication execution mechanism is a perfect match for this requirement.

Predicated execution, or simply predication, is a mechanism that supports conditional execution of individual operations based on a Boolean guard or predicate [8][16]. Predicated operations are fetched regardless of their predicate value. Operations whose predicate is TRUE are executed normally. Conversely, operations whose predicate is FALSE are nullified, and thus are prevented from modifying the processor state. With predication, the representation of programmatic control flow is inherently changed. A processor with predication can support conditional execution either via conventional branches or with conditional operations. As a result,

the compiler has the opportunity to physically restructure the program control flow into a more efficient and parallel form for execution on an ILP processor. It is the effective integration of predication with the EPIC execution microarchitecture that provides an efficient method of disabling memory instruction requests.

In [11], the benefit of predication support was studied with a predication compiler. However, the accompanying compilation model was based solely on synthesizing predicated code based on the original control flow. One fundamental problem with using traditional compiler techniques to deploy predicated execution is that existing compiler techniques do not significantly alter the program's original control and data flow structure to sufficiently use the full potential of the predicated execution mechanism. For modern compilers to expose sufficient amounts of instruction level parallelism, techniques for analyzing the original program control flow and data flow into new EPIC-aware forms must be developed. This paper illustrates new compiler transformations that exploit value locality characteristics and commonly occurring data dependence program structures to create more memory efficient code. Such execution paths can be strategically synthesized to increase the amount of instruction level parallelism that can be efficiently executed on a predicated architecture. Overall, this research proposes new compiler transformations whose projected impact is to extend the benefits of predicated execution in the EPIC architectures.

2. MOTIVATION AND BACKGROUND

Currently memory performance issues are limiting the performance of EPIC architecture implementations. It is estimated that nearly 50% of the Intel/HP Itanium execution time on the SPEC benchmark suite is spent stalled waiting for both instruction and data cache misses. Figure 1 illustrates the data cache hit rate and the percentage of cache-miss stall cycles relative to the overall execution for a few SPEC benchmarks evaluated on an EPIC architecture. It is important to note that although the benchmarks achieve nearly an average of 90% hit rate, 40% of EPIC execution is dominated by memory system stalls of the process. Although out-of-order processor systems can adapt to cache misses by executing other instructions surrounding the load, this is not the case for EPIC machines. For an EPIC in-order design, the execution pipeline stalls when another instruction attempts to use the result of a load that has missed in the cache. As this execution semantic is a substantial performance bottleneck, it is critical to improve memory efficiency by eliminating unnecessary demands on memory system. The inherent problem requires substantial analysis in synthesizing code with more efficient memory access patterns and directing new compiler-directed prefetch mechanisms. To improve memory system efficiency, the compiler will need to play a more active role in analyzing and transforming the original data flow structure of a program.

There are a number of advantages to an EPIC architecture's predicated execution support than simply reducing branch mispredictions and enabling the parallel execution of multiple control flow paths. To date, very little research has been performed to investigate radical changes to a program's original data flow and control flow. This paper outlines methods of using predicated execution to substantially improve the

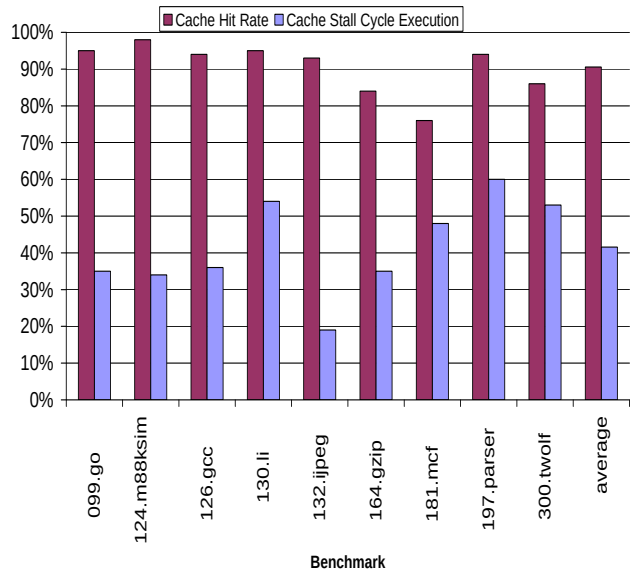


Figure 1: Cache hit rate performance and distribution of processor cycles spent on cache misses.

Itanium's memory access efficiency. Value locality, the repetition of computation values may serve to reduce necessary memory operations of a program. Although new architecture attempts have been proposed to simply reuse previous results of execution [18] based on value locality, there are more immediate benefits that can be made in the Itanium architecture using predicated execution.

By analyzing data dependence graphs with value locality profile information, several dependence graph structures can be transformed to eliminate the number of necessary memory accesses. In a dependence graph there are often disjunctive and conjunctive functions applied at the join point of two sequences of dependent operations. Consider a dependence graph with two halves, section A and section B. Certain values generated by section A of the graph will render the operations on the section B unnecessary. If load operations are present on section B of the dependence graph, they also are unnecessary. With the in-order nature of an EPIC architecture, the section B loads can be predicated based on finding the run-time value in the first dependence chain. In the case that newly predicated loads would miss in the cache, their predicate operands will indicate that it is unnecessary to access the memory location.

As a second example, consider the simple set of integer operations displayed in Figure 2 that may be found in a loop structure. In this case, it is not necessary to perform the loads of array elements B[i] and C[i], if the result of A[i] is zero. In this case, zero multiplied by anything will remain zero. By predicating loads to array B and C are predicated based on the respective value of array A, many unnecessary cache misses are eliminated. Similarly, silent store (store operations writing the same value to memory) can also be eliminated. Overall, there are two beneficial effects of eliminating unnecessary memory accesses: elimination of memory latency and elimination of cache pollution. As such, the proposed approach may have substantial effects in aiding

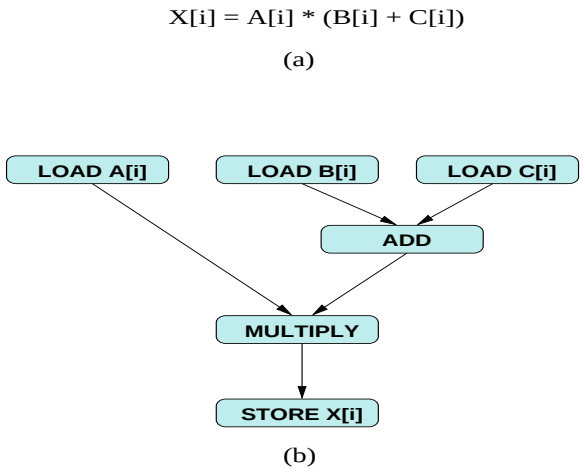


Figure 2: Code example of value-irrelevant memory accesses (a) source code and (b) dependence graph.

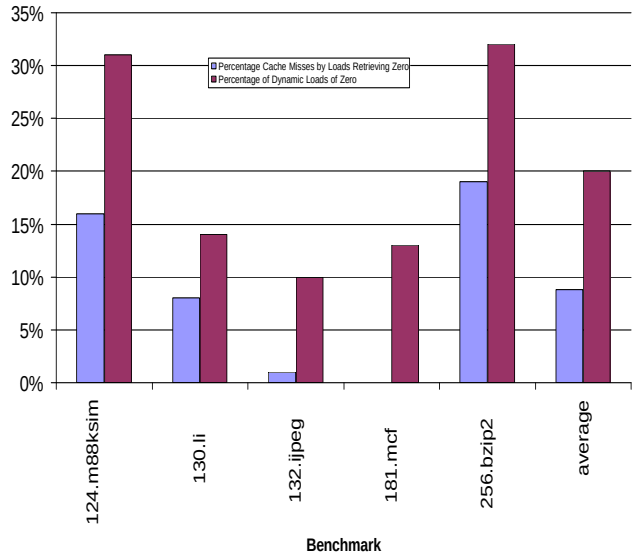


Figure 3: Value locality characteristics of dynamic load execution.

the Itanium architecture's performance.

Figure 3 illustrates the degree to which zero valued loads contribute to cache inefficiency. Cache simulations for this set of SPEC Benchmarks revealed that load operations which evaluate to zero occur quite frequently. Furthermore a significant set of these loads are responsible for data cache misses. Since the occurrence of zero valued loads is relatively high, it appears that cache efficiency could be increased by applying value locality optimization techniques. Based upon this information, our proposed work is to use predicate instructions to eliminate the execution of memory operations which could be removed due to their data dependence on zero valued functions.

According to the IMPACT EPIC cache simulations which were run using the IMPACT compiler suite, load operations that evaluated to 0 for more than 95% of their executions, contributed as much as 20% to the total data cache misses. Although the distribution of the zero valued load operations are varied over the benchmarks, the presence of such high percentages beckons the use of our proposed optimization. *132.ijpeg* and *181.mcf* appear to be the only benchmarks that did not exhibit a bias toward zero valued loads. The others showed significant contribuxtions to data cache misses from these operations.

2.1 Background and Related Work

Related Work. There has been significant previous research work in the area of program reformulation. In addition to the arithmetic reformulation work mentioned in the previous section, a large body of related work in the area of control flow optimization has also been carried out. These methods can be classified into three major categories: branch elimination, branch reordering, and control height reduction. Branch elimination techniques identify and remove those branches whose direction is known at compile time. The simplest form of branch elimination is loop unrolling, in which instances of back-edge branches are removed by replicating the body of the loop. More sophisticated techniques examine program control flow and data flow simultaneously to identify correlations among branches [4], [14]. When a correlation is detected, a branch direction is determinable by the compiler along one or more paths, and the branch can be eliminated. In [14], an algorithm is developed to identify correlations and to perform the necessary code replication to remove branches within a local scope. This approach is generalized and extended to the program-level scope in [4]. The second category of control flow optimization work is branch reordering.

Another category of program reformulation is control flow optimization research, which focus on the reduction of control dependence height. This work attempts to collapse the sequential evaluation of linear chains of branches in order to reduce the height of program critical paths [17]. In an approach analogous to a carry look-ahead adder, a look-ahead branch is used to calculate the taken condition of a series of branches in a parallel form. Subsequent operations dependent on any of the branches in the series need only to wait for the look-ahead branch to complete. The control dependence height of the branch series is thus reduced to that of a single branch. The mechanisms introduced herein also serve to reduce control dependence height. This chapter, however, leverages an approach [3] to minimization and re-expression of control flow networks that is far more general than those proposed in previous work.

The final category of related work is value-based compiler transformations. Value-based optimization include all optimization based on a predictable value or range of values for a variable or instruction at run time. These include constant propagation, code specialization [7], optimization assuming the value predictability of an instruction, continuous optimization, and partial evaluation. Variables and instructions that have invariant or predictable values at run time, but cannot be identified as such using compiler analysis, can benefit from value-based compiler optimization.

Predicated Execution Background.

Predicated execution is a mechanism that facilitates the conditional execution of individual instructions. Predication is most commonly utilized in a compiler by employing *if-conversion*. If-conversion is a process whereby conditional branches are converted into predicate defining operations, and operations along alternative paths of each branch are guarded under the computed predicates [1][15][12]. Predicates are registers that store a single bit value, representing either TRUE or FALSE. Each instruction is associated with a particular predicate, known as its guard predicate, that determines its execution.

With if-conversion, complex nets of branching code can be replaced by a straight-line sequence of predicated code. There are two major benefits associated with applying if-conversion. First, a compiler can eliminate problematic branches from the program. In doing so, all the associated overhead with these branches is removed, including misprediction penalties, penalties for redirecting sequential instruction fetch, and branch resource contention [10]. Second, predication facilitates increased ILP by allowing separate control flow paths to be overlapped and simultaneously executed in a single thread of control.

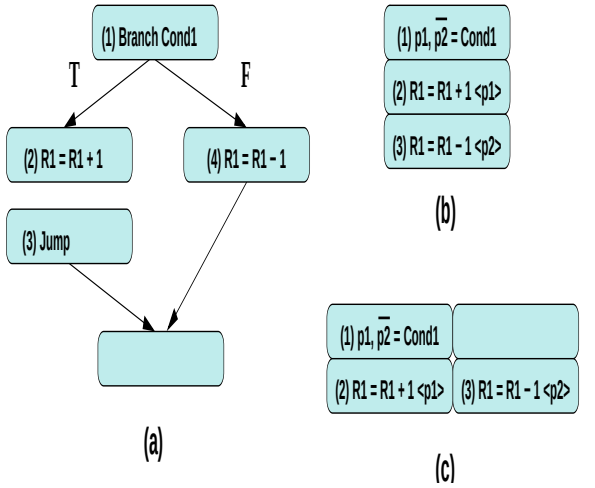


Figure 4: An example code segment (a), after if-conversion (b), and after scheduling (c).

Figure 4 contains a simple example illustrating the concept of predicated execution. Figure 4(a) shows sequential if-then-else constructs, called *hammocks*. The branch outcomes are determined by the evaluation of the branch condition $Cond1$. Depending on the outcome of the first branch register, $r1$ is either incremented or decremented. Figure 4(b) shows the code segment after if-conversion, and Figure 4(c) shows the code after scheduling. Here the two branch conditions have been transformed into comparison instructions that define predicate destinations. The example illustrates how support for predicated execution allows the multiple path contexts to be executed in parallel on a wide-issue machine.

3. APPROACH

Motivated by the potential of aggressive techniques for transforming arithmetic expressions, this section introduces a new approach to optimizing program structures. The goal of this work is to develop a systematic methodology for reformulating specific regions of program data flow for more efficient exploitation of their inherent value locality characteristics on an ILP processor. Dynamic value locality information is proposed to be gathered and then represented as a *dynamic value dependence graph*. A new, more parallel computation is synthesized with the goal of reducing dependence height. To accomplish the desired optimization and synthesis, the parallel computation is modeled as new control flow structures which are added through a proposed process known as if-insertion. In turn the control structures are converted to a predicate representation using the traditional if-conversion technique.

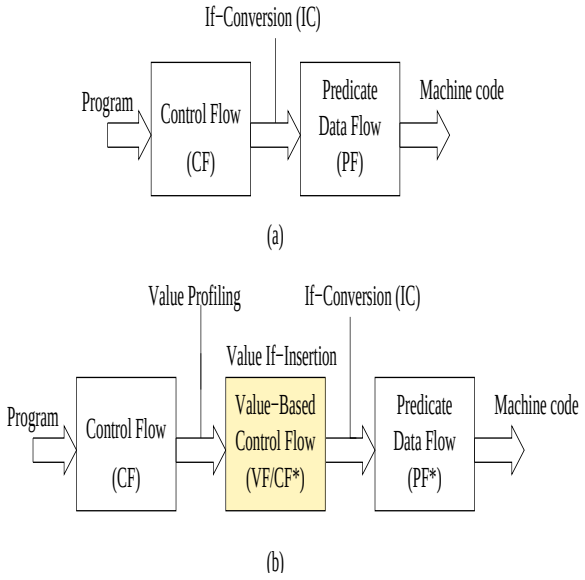


Figure 5: Program transformation paths.

Figure 5 illustrates the program transformation paths and compiler techniques presented in this section. There are two illustrated program representation paths: traditional control flow (CF), to predicate data flow (PF) using if-conversion, and traditional control flow (CF) to if-inserted value-flow (VF) to predicate data flow (PF). The predicate representation of code effectively designates the execution conditions of the newly created program sequences. The key idea behind the value-based optimization is to synthesize optimized code structures by analyzing the dynamic profile-determined program value characteristics and applying a series of control flow and data flow transformations that use both a predicate-based compiler representation and predicated execution microarchitecture support.

Two domains of transformation techniques are proposed: predicate-based data-irrelevant and control-irrelevant memory elimination transformations. The transformations are guided with value profiling [6] and cache profiling information gathered during training runs of selected benchmarks. The profiling information is used to generate new control instructions within the code (if-insertion) which are then

collapsed into predicate-based data dependent instruction using if-conversion.

Eliminating Data-Irrelevant Memory Accesses.

The data dependence graph of a particular computation may compute the same result by only executing a subset of the graph's execution nodes. For example, Figure 6 illustrates an example from *126.gcc* of memory operations that become irrelevant to code's result when some instructions resolve to the value zero. The source code of Figure 6(a) updates two arrays (basic_block_live_at_start and basic_block_live_at_end) being updated based on the result of a logical-or-and sequence with their original values. Specifically this code is performing dataflow analysis within a compiler and calculating the live range of a set of program variables. The loop is invoked several times, yet the live variable information is very sparse (zero-valued loads) and change infrequently. The execution is unnecessary when the logical-and operation result is zero and any cache misses are unnecessary.

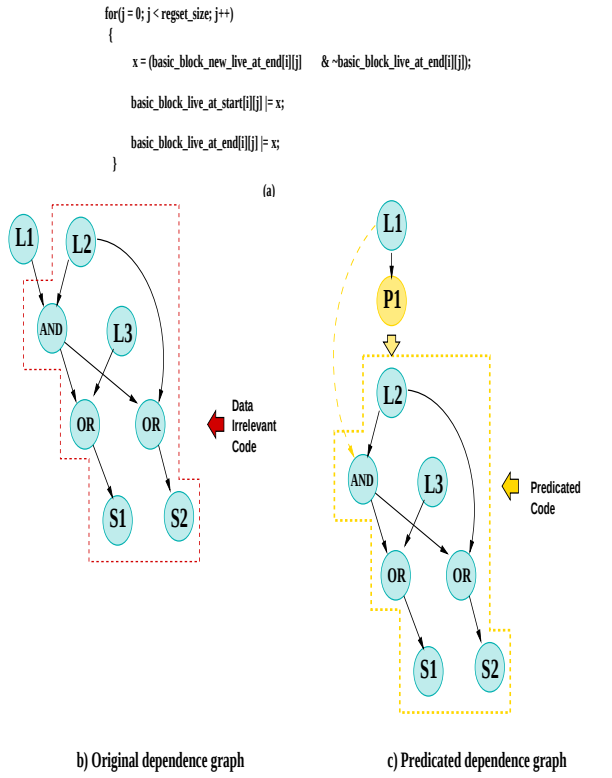


Figure 6: An example from *126.gcc* of data irrelevant source code (a), after if-insertion (b), and after if-conversion (c).

Figure 6(b) illustrates the dependence graph of the *126.gcc* example and highlights the data irrelevant code that results when a triggering load of basic_block_new_live_at_end is zero. Figure 6(b) demonstrates the predicate-based value optimization technique to reduce the execution of the unnecessary memory instructions. The instruction **P1** is the predicate definition instruction that controls the selected data-irrelevant code and is based on comparing a loaded result to the value zero. Through experimental evaluation of the predicated version of the *126.gcc* code, several thousand simulated stall cycles were eliminated.

Eliminating Control-Irrelevant Memory Accesses.

Figure 7 shows an example of Control-Irrelevant memory guarding. Figure 6(a) shows a code block with a branch followed by a load-add pair. Speculative placement of the load moves the load above the branch. Because the add is dependent upon the load, the add can also be speculated above the branch in Figure 7(b). Speculating the load allows for increased parallelism through code reordering. The problem with speculation is that the load is executed every time that particular section of code is entered, regardless of whether or not the branch takes. Because the load is irrelevant if the branch takes, it is Control-Irrelevant. There are two main problems with Control-Irrelevant loads. The first is the cache pollution caused by loading a value that isn't used or needed. The second is important for in-order machines. If the speculative load misses in the cache the machine must stall on the speculative add instruction. If the branch takes the machine stalled for no reason. This is an obvious performance deterrent that must be addressed. We propose a predicate based method called Control-Irrelevant Memory Guarding. For the case in Figure 7 at the time the load is executing the condition determining whether or not the branch is taken is known. Using predication, we can predicate the speculative instructions so that they only execute if the branch is not taken Figure 7(c). By using predication to guard the speculative instructions, we can achieve the benefits of speculation without the drawbacks of irrelevant execution.

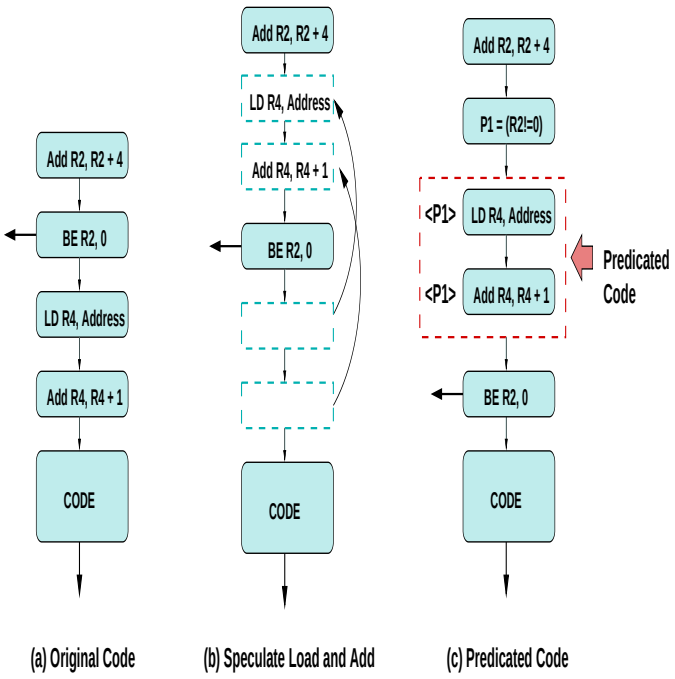


Figure 7: An example of potential control irrelevant load segment (a), after if-insertion (b), and after if-conversion (c).

4. EXPERIMENTAL EVALUATION

4.1 Methodology

The IMPACT compiler and emulation-driven simulator were enhanced to support a model of an IMPACT EPIC architecture [2] and simulation of the code transformation techniques respectively introduced in Section 3. The benchmarks used in all experiments consist of SPECINT95 and SPECINT2000 programs. The base level of code consists of the best code generated by the IMPACT compiler, employing function inlining, superblock formation, and loop unrolling.

The base processor modeled can issue in-order six operations up to the limit of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. The instruction latencies used match the Itanium microprocessor (integer operations have 1-cycle latency, and load operations have 2-cycle latency.) The execution time for each benchmark was obtained using detailed cycle-level simulation. The parameters for the processor include separate 32K direct-mapped instruction and data caches with 32-byte cache lines with a miss penalty of 12 cycles. A 1M second-level unified cache (with memory latency of 100 cycles) is also simulated. For branch prediction, a 4K entry BTB with two-level correlation prediction with a branch misprediction penalty of eight cycles is modeled.

4.2 Results and Analysis

Figure 8 illustrates the percentage of cache missing loads that are caused by a speculated memory operation. On average, approximately 44% of loads that miss in the cache are speculative. Preliminary analysis shows that approximately 1/3 of the speculated loads prove to be control irrelevant. This means that 1/3 of the loads in Figure 8 do not need to be executed. Speculative loads allow for increased parallelism, however, this data shows that speculation can also have negative effects in the form of irrelevant execution. Irrelevant execution of loads causes cache pollution and possible stalls for speculated data that is not needed. This indicates a high potential for optimization. By using predication to guard speculative loads from control irrelevancy, the benefits of speculation can be achieved without the penalty of irrelevant execution.

Figure 9 displays the opportunity for removal of control irrelevant loads via predication. Of the speculative, control irrelevant loads that cause a miss, Figure 9 displays the percentage that could be removed through the proposed predication techniques. Compiler analysis indicates that on average 60% of irrelevant speculative loads could be guarded against irrelevant execution. This indicates a large number of candidates for predicate based optimization.

The performance speedups of Figure 10 indicate that all benchmarks tested improved markedly with the proposed compiler based predicate transformations. The first data set compares the data irrelevant code to the base case of superscalar optimized code. It is noteworthy that the data dependence removal proves beneficial on code that has already passed an aggressively optimizing compiler. Similarly, the results from the second data set demonstrate that the removal of data dependence still boosts performance when run with the classic control removal predicate instructions. Benchmark *124.m88ksim* exhibited the greatest speed up, 6% for both cases. This was expected due to the high percentage of zero valued load instructions, 31%, shown in Figure 3. While *130.li* only improved by 3%, which again reflects a smaller percentage of zero valued loads during execution, 14%. Cache performance evaluations indicate that

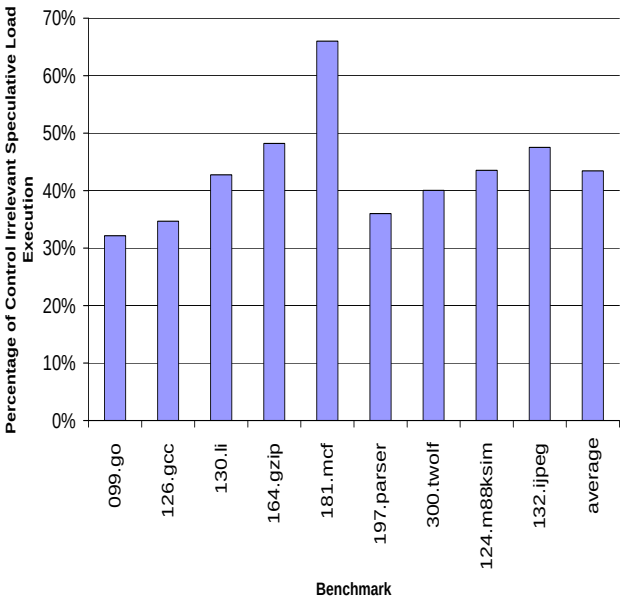


Figure 8: Distribution of cache misses caused by speculative loads.

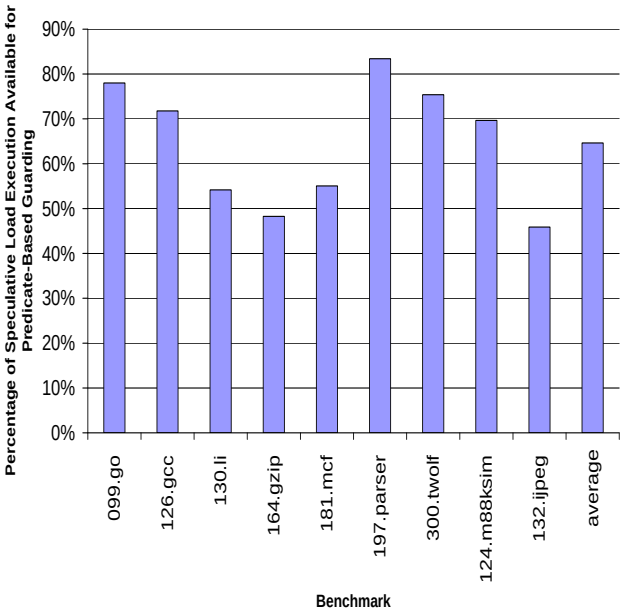


Figure 9: Potential speculative load execution guarded with predication.

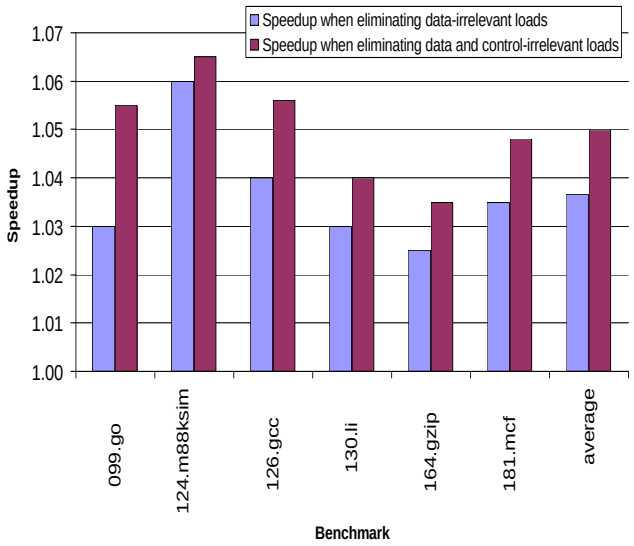


Figure 10: Performance improvement.

the performance is achieved by eliminating between 3-5% of the original cache misses.

5. SUMMARY

Current advanced compilation techniques do little to alter a program's original control flow. Because of this, modern optimization can do little to reduce the redundancies and irrelevant execution so common in today's programs. Through the use of profile guided predication, new methods were developed to alter the inherent control flow of a routine, resulting in increased performance and elimination of redundant execution. By changing the data and control flow through predication, improved memory access efficiency is achieved. Because current predicated architectures are implemented on in-order machines, the memory system's performance is vital. Using predication to remove data and control irrelevant loads results in a more pollution-free memory system as well as performance benefits resulting from the elimination of unnecessary stalls. Preliminary experiments show increased performance with relatively simple predication optimization.

The techniques presented in this paper are confined to the scope of compilation trace of code. Although interesting results were established with the system, future work will investigate methods of determining whether many other memory accesses are necessary to a program's execution. Such a technique will require a global analysis and optimization infrastructure.

6. REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu. The program decision logic approach to predicated execution. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 208–219, May 1999.

[4] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 146–158, June 1997.

[5] R. Bodik, R. Gupta, and M. L. Soffa. Complete removal of redundant computation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 259–269, December 1997.

[7] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *The Journal of Instruction-Level Parallelism*, 1, March 1999. http://www.jilp.org/vol1.

[8] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[9] T. L. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[10] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 217–227, December 1994.

[11] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, June 1995.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[13] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct 1992.

[14] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 55–66, June 1995.

[15] J. C. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[16] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[17] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 57–69, December 1995.

[18] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 205–215, December 1998.

EPIC Instruction Scheduling Based on Optimal Approaches

Steve Haga
University of Maryland
College Park, MD
stevhaga@eng.umd.edu

Rajeev Barua
University of Maryland
College Park, MD
barua@eng.umd.edu

ABSTRACT

This paper presents a method for instruction scheduling that considers the scheduling restrictions inherent in VLIW processors, particularly EPIC. EPIC imposes restrictions on the nature and code-order of instructions that may issue in the same cycle. To express these restrictions, the instructions are grouped into instruction classes such as arithmetic, memory, floating point and branch instructions. Allowable ordered combinations of instruction types are called *templates*.

Most existing methods for instruction scheduling do not consider templates except at their last stage. For example, trace scheduling and similar methods focus on moving instructions across basic blocks. After movement, the instructions are usually put in templates using greedy algorithms such as list scheduling. There is a significant opportunity to improve performance if algorithms superior to greedy are used for template scheduling.

The scheduling method in this paper takes the following approach. It begins with a provably optimal algorithm for template scheduling. This algorithm is not feasible, however, since it results in an exponential compile time. Two classes of methods are used therefore to drastically reduce the compile time. First, aggressive branch-and-bound techniques are used to prune portions of the search space while retaining the optimality guarantee. Second, non-optimal heuristics are used to guide the search towards more promising solutions quickly. Results show that our techniques are able to reduce the number of NOPs in integer programs by 56% on average compared to the list-scheduling based greedy algorithm in the SGICC compiler, with only a 17% increase in compile time. Floating point programs see a 35% reduction in NOPs with a 22% increase in compile time.

Keywords

EPIC, templates, instruction scheduling, IA-64

EPIC 1 '01 Austin, Texas USA

1. INTRODUCTION

This paper presents a new approach for scheduling the instructions of a basic block in VLIW machines that impose constraints on instruction placement. These constraints are represented by a set of allowable *templates*. Each template specifies one allowed sequence of instruction types (integer ALU, branch, floating point, etc.). Fixed-width VLIWs employ a single template, while EPIC architectures [15, 2] allow instructions in multiple templates to execute simultaneously. Code scheduling under template constraints is an important but difficult problem because poor schedules require many extra NOP instructions. Adding NOPs wastes resources, impacting performance, power, and memory requirements.

The approach described contains several innovative optimizations that improve code density, where code density is defined as the ratio of the number of useful instructions to total instructions, where total instructions include useful instructions and NOPs. First, we present a provably optimal algorithm to perform instruction scheduling. Second, since the optimal algorithm is exponential time, *branch and bound techniques* [5] are used to drastically reduce the compile time; these techniques make use of a novel upper bound for the code density of a basic block. Third, an innovative, constant-time method for optimally selecting templates for a set of parallel instructions is presented. Fourth, non-optimal heuristics are applied to the larger basic blocks, where optimal scheduling may take too long.

The success of these techniques is shown in the results. For integer benchmarks, 56% of NOPs are removed, as compared to the SGICC compiler for the IA-64 [14, 1], at a cost of 17% more compile time. In floating point benchmarks, 35% of NOPs are removed, as compared to the existing compiler, at a cost of 22% more compile time. Our results are produced by modifying the SGICC compiler.

A new instruction scheduling method is needed for EPICs because traditional instruction scheduling algorithms are ineffective for template-constrained systems. Well known techniques based on trace-scheduling [7, 12, 11, 4] focus on ways to move instructions across basic blocks, but say nothing on how to assign and fill templates. Instruction scheduling, an NP-complete problem [13], is usually done either by *list scheduling* [13], or by some technique which considers the resource constraints of the machine. No current method considers the template restrictions of EPICs prior to the ordering of instructions; thus, extra NOPs may be produced. The cause is that template choices are correlated: choosing to schedule an instruction limits the set of potential templates available to for the next choice. Further, the choice

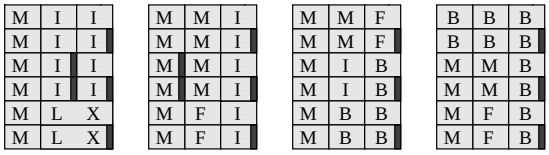


Figure 1: A list of all available templates for IA-64. Dark vertical bars represent stop bits.

of instructions to schedule on this cycle can easily affect the quality of the solution for scheduling on the next cycle.

An outline of the rest of the paper is as follows. In section 2, EPIC and IA-64 are described. In section 3 is a motivational example of the dangers of not considering templates when ordering instructions is presented. In section 4, related works are examined. In section 5, optimal approaches are considered. These include schedules for an instruction group, as well as an entire basic block. Branch and bound techniques are presented, to reduce compile-time. In section 6, non-optimal heuristics to further improve run time are presented. In section 7, the performance of the approach is evaluated. In section 8, conclusions are presented.

2. EPIC TEMPLATES

EPIC [15] architectures are a class of VLIWs that introduce many new features, including variable-length parallelism. While much of our approach is applicable to any VLIW, our research focuses on EPICs, where variable-length parallelism is achieved by the mechanism of *stop bits*. Conceptually, after every instruction there is a stop bit that may be set. These stop bits form the boundaries of parallelism; a sequence of instructions without any set stop bits between them are said to belong to the same *instruction group*. EPIC templates specify the stop bit placement, as well as the allowed combinations of functional unit types. A set of instructions that have been mapped onto a template are called a *bundle*. The bundle does not represent one instruction group. Instead, the stop bits indicate the instruction groups, and there may be several stop bits inside of one bundle, or several bundles between stop bits.

Since our algorithms are tested for IA-64 [9], we now consider the IA-64 ISA, the EPIC architecture of greatest practical interest. IA-64 uses bundles of three instructions and provides 24 templates, as shown in figure 1. It defines four functional units: Integer, Memory, Floating Point, and Branch; and six instruction types. Four of these types (I, M, F and B) indicate instructions which must be executed on the corresponding functional unit. The two other instruction types are A (which may be called a *super type* because it may execute on either an I or an M) and LX (which fills 2 bundle slots and uses the I and B functional units). The current implementation of IA-64, Itanium [10], imposes additional constraints; when an instruction group violates these constraints, Itanium splits it into two cycles.

3. INSTRUCTION ORDERING WITHOUT REGARD FOR TEMPLATES

An example is now presented, which highlights the need to consider templates when scheduling instructions. Current basic block scheduling methods do not consider templates until after assigning instruction groups. Of these methods, list scheduling is the simplest to describe; and it is the approach used by SGICC– the compiler we use and compare results against. Therefore we consider list scheduling in the following example. The resulting schedule, however, with its poor fit to the templates, is typical of all current methods.

Consider the example Directed Acyclic Graph, DAG, of figure 2(a), showing insructions and their dependencies. List scheduling selects one instruction out of those that are ready to schedule *i.e.,* whose dependencies are met, based on a user-specified priority function; for SGICC, it is the height in the dependence DAG. Figure 2(b) shows that, with the wrong heuristic, list scheduling produces three NOPs for six instructions, using the IA-64 templates, where as the optimal schedule in figure 2(c) has no NOPs at all.

List scheduling results in figure 2 as follows. In the first cycle, there are two ready instructions: M1 and I2. List scheduling sees that M1 is used earlier and schedules it first. Then it must decide whether to end the instruction group or to schedule I2 also; it schedules I2 to achieve parallelism. Having no more possible instructions, a stop bit is inserted. The dependencies are updated and there are now three ready instructions: M3, M4, and M5. M3 has the earliest deadline, and so is scheduled next. It is an M-type instruction, where as the the IA-64 templates are such that a stop bit after the second slot forces the third slot to be an I type. Therefore the first bundle is given a NOP, and a new template is begun for M3. M4 is scheduled next. Then M5 is tried. since it does not fit into the template, that template's last slot is filled by a NOP. It decides to delay M5. On the next cycle M5 and F6 are both ready and both can be scheduled. The optimal solution shown in figure 2(c) uses one less template.

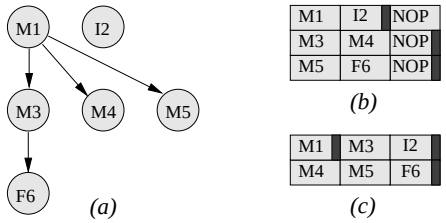


Figure 2: Scheduling example. (a) the DAG (b) the result of scheduling without considering templates: (MI;I MMI; MFI;) (c) an optimal schedule: (M;MI; MMF;)

4. RELATED WORK

The term instruction scheduling is often applied to methods which assign basic blocks to instructions; these methods are not to be confused with instruction scheduling within a basic block, such as in our research. The purpose of these basic-block assigning techniques is usually to reduce the run time by moving code, when possible, into less frequently used blocks, or to perform if-conversion, so as to increase the available parallelism. One technique, *trace scheduling* [7], constructs *traces*, which are identify as a sequence of basic blocks where the next block in the sequence is a successor of the current block, and where there is a very high probability that if the first block is entered, the entire trace is executed. Since the probability of this entire path occurring is high, the trace may be optimized, even if it causes less-frequent blocks

outside of the trace to run more slowly. Computation that is not used in the trace is attempted to be moved outside of the trace, a form of instruction scheduling, across basic blocks. *Superblocking* [12] and *hyperblocking* [11] are techniques based on trace scheduling. An alternative approach, *wavefront scheduling* [4], is suited for machines that allow speculation and predication, such as EPIC. It must be understood that all of these methods are performed prior to instruction scheduling within basic blocks. Our technique is not meant to replace these existing methods, but to work with them. We later describe how our method may, in fact, be integrated with trace-scheduling based methods.

Concerning scheduling instructions within a basic block: list scheduling, described in section 3, has proven sufficient for superscalars– they have a reorder buffer to correct bad schedules and no template constraints. VLIWs have motivated more advanced techniques that consider the resource constraints of the system. One such technique, [8], which targets embedded processors, creates a DAG of pairwise edges that represent instructions which are mutually schedulable. Instruction scheduling is then performed by finding a minimum-cost clique covering for the DAG. Another possible approach, [3], schedules instructions based on Finite State Automata methods. The dependencies of each instruction are represented as a bit-vector. If the AND-ing of two bit vectors yields an empty vector, then the two corresponding instructions do not share any resources, and may be scheduled together. Given that a particular set of instructions has been scheduled in a certain cycle, the OR-ing of their bit vectors represents the resources currently used. To construct the finite state automata, a bit vector of resources used by a set of instructions represents a state, and choosing to schedule an additional instruction on this cycle represents a transition to a new state.

Neither of these methods address templates, nor may they be easily modified to this end, for the following three reasons. First templates impose variable resource restrictions, where as the methods of [8],[3] rely upon the constant resources available in fixed-width VLIWs. Instruction groups may begin in different slot positions, resulting in different resources constraints. For instance, the issue window of Itanium is six instructions if the instruction group begins in slot the first slot, but is only four instructions, if the group begins in the third slot. Second, the methods of [8],[3] only consider whether individual pairs of instructions are schedulable, where as template restrictions apply to the entire instruction group. For example, in an Itanium instruction group, not more than $\frac{1}{3}$ of the instructions may be of F type and not more than $\frac{2}{3}$ of the instructions may be of I type (from figure 1). Because the template constraints are relative to the entire instruction group, comparisons of instruction pairs are insufficient. Third, these template constraints are loose. NOP instructions may be used to meet the constraints. And satisfying all of the relative constraints does not guarantee a match to the templates. for instance, an instruction group with one F and two I types meets the IA-64 constraints, but there is no IIF template.

In [6] a template-aware method of instruction scheduling for the Itanium implementation of IA-64 is presented, based on Integer Linear Programming (ILP) and dynamic programming. The resource constraints of the Itanium processor are formulated into an ILP problem, which is solvable by commercial software. The solution of the ILP problem identifies the instruction groups of the final code, but does not consider templates. Fitting these instruction groups into templates is not simple, because the template selection depends on the starting slot position for the instruction group. To overcome this difficulty, the solutions for each possible starting and ending slot position are found. Dynamic programming then determines the globally best schedule. This approach often uses the provably fewest number of execution cycles for an Itanium. This claim is not strongly supported, however, since it depends on the validity of the conditions of the ILP model; one such condition is that the Itanium's issue width is 6 instructions, when in fact, the issue width is less for some starting slot positions. While they found that resource violations are uncommon, their benchmarks do not include floating point or highly parallel code, which are more likely to cause such a violation– an instruction group of just two floating types has a violation if not begun in slot 0, and an instruction group of 5 or 6 instructions may not fit into Itanium's variable-width issue window.

There are four fundamental differences in the problem being solved by our work and that solved in [6]. First, we improve the code density and the execution time, while [6] primarily improves the execution time, and secondarily, the code density. Second, we accomplish this by considering the templates at every step of instruction scheduling; in [6], templates are not considered until after instruction groups are assigned. Thus the situation illustrated in figure 2(b) is still possible with [6], since the schedule of figure 2(b) does use the minimum number of cycles, despite extra NOPs. Indeed, [6] does not claim to minimize the number of NOPs, but only the number of execution cycles. Third, our approach currently considers the general IA-64 instruction scheduling problem (*i.e.,* no issue width limits), but [6] considers the highly-constrained Itanium implementation. Indeed, if [6] is modified to solve our general problem, it simply chooses some random partitioning of instruction groups which uses the fewest number of cycles; in contrast, our technique also chooses a partitioning of instruction groups which executes in the fewest number of cycles– but not randomly, rather, so as to minimize the number of NOPs. Fourth, the extension of our approach to consider Itanium's limitations, as discussed in section 5.3, is straight-forward and would guarantee a minimum cycle solution, as well as a minimum number of NOPs; however, it is not clear how template considerations could be formulated as an ILP problem such as in [6]. Thus our methodology, when modified to consider real-machine constraints will also guarantee a minimum number of cycles– and a minimum number of templates as well, which [6] cannot be modified to do. The fact that our approach, in its current form, is not bound to implementation constraints, has the positive benefit that we can study the general IA-64 problem, such as what portion of NOPs is imposed by the ISA, regardless of the implementation constraints, allowing us to uncover what properties of the IA-64 templates account for this behavior.

Our work and [6] are not in conflict. If it is found that their approach is quicker (a decision which requires more performance data on their technique), it could be employed first, and our technique only employed for those basic blocks for which [6] yields low code density.

5. OPTIMAL BASE ALGORITHM

In this section, we consider optimal methods to schedule

a basic block. In section 5.1, we examine what properties an EPIC must have in order to be applicable to our methods. In section 5.2, we develop an optimal algorithm to select the templates for one instruction group. In section 5.3, we present a recursive search algorithm to find the best schedule of a basic block, by means of trying all possibilities. In section 5.4, we improve the compile-time of the search algorithm, through branch and bound techniques. In order to further reduce compile times, section 6 considers non-optimal heuristics.

5.1 Assumptions

We first discuss our optimal base algorithm, which begins with certain assumptions. It is assumed that whatever EPIC architecture is being considered has the property of *paired templates*. Templates are paired if, for each stop-bit contained within any template, there is another template which is identical, except that it does not have that stop bit. The IA-64 is an EPIC which has paired templates, as can be seen by examination of table 1. Assuming paired templates, we arrive at an obvious but important optimization which is used in our algorithm:

THEOREM 1 (OPTIMALITY WITHOUT SPLITTING). *If all templates are paired, then the code density of an optimally scheduled instruction group never improves by dividing that instruction group into two instruction groups, separated by a stop bit, and optimally scheduling each.*

PROOF. Suppose an instruction group, Z is partitioned into two instruction groups, A and B. A is optimally scheduled into the available templates, and a stop bit is inserted after the last instruction of A. B is now optimally scheduled, starting at the first slot after the stop bit, if it occurs in the middle of a template. The resulting schedule may be written as "$A;B;$", where stop bits are indicated by a ";".

Regardless of how many instructions are contained in A or B, there is only one template containing the ";" that lies between A and B. Replacing this template by its template pair yields the schedule "$AB;$", which contains only valid templates, since "A;B;" contained valid templates and only template has been changed, and that in a valid way. Since "$AB;$" has no internal stop bits, it represent a single instruction group, and since A and B partitioned Z, "$AB;$" represents a valid schedule of Z. There may exist even better schedules for Z, but at least we have identified one schedule which is as good as the best possible solution with partitioning. Thus, partitioning an instruction group never reduces the number of bundles needed to optimally schedule it. □

It is desirable to not sacrifice execution time in the interest of code density, which is facilitated by our assumption of a machine without constraints. Since we currently target a general IA-64 machine under the assumption that instruction groups of arbitrary size issue within a single cycle, a solution which requires the minimum number of cycles may be guaranteed if all instructions are scheduled by their deadlines. Since only minimum cycle solutions are considered, the best measure of the quality of a schedule is its code density. Extension of the algorithm to real machines is discussed in section 5.3.

5.2 Instruction Group Template Selection

For the first time, we show that certain EPICs have properties that allow for constant-time, *i.e.,* O(1), selection of optimal templates for an instruction group. This is a noteworthy, in that one might expect that the list of instructions must be traversed in order to be scheduled, in order to place them into templates. O(1) time is achievable, nonethe-less, because it is found that optimal template selection may be performed without checking each instruction. All that is required is to know the *number* of instructions of each instruction type. If these numbers are not known, the additional cost of counting these is linear; however, for the bounding methods based on constant-time template selection, which are presented later, there is no need to count the instructions, and the algorithm is truly constant time.

Three properties are identified, which an EPIC must have in order to be guaranteed to have optimal template selection in constant time. First, a deterministic template choice for bundles with an internal stop bit. When an instruction group finishes in the middle of a bundle, the subsequent instruction group begins in the next slot of that bundle, which is to say that a template with an internal stop bit is used. A deterministic template choice means that, given the instructions already scheduled into the bundle prior to the stop bit, there is only one choice for the remaining instruction types of the bundle; otherwise, different algorithms would be needed for each starting slot. From table 1, it is seen that IA-64 has only one template pair for each internal stop bit location. Second, super types must not partially overlap. For instance IA-64 has one super type, 'A', which represents those instructions that may be scheduled into either an M or an I functional unit. An overlapping super type (which would prevent quick selection if it existed) could be for instructions that schedule in either the I or F functional units; it overlaps because it also maps to I. When super types do not partially overlap, there is never ambiguity about which type to choose. Third, ideally the architecture should not allow intra-dependencies within an instruction group; otherwise the instructions may not be schedulable to the chosen templates. IA-64 and most architectures break this condition. It is only a technical condition, however, since no instance is found in our testing, where the instructions could not be mapped to the chosen templates. For the remaining, we loosely speak of IA-64 as meeting the above criterion.

The template selection algorithm, as it pertains to IA-64, is shown in table 1 and is now briefly described. It takes as input: *slot*, the starting slot position for this instruction group, which depends on where the previous instruction group finishes, and $TypeCnt$, an array which contains the counts of every instruction type within the instruction group. If the previous instruction group did not end on a bundle boundary, the selection begins by filling in the remaining slots of the first bundle according to the predetermined template. Slots are filled by choosing and removing instruction types from $TypeCnt$, which is accomplished by the DEC function, to be described momentarily. Once this first bundle is complete, the basic approach is to first remove the types with the fewest choices. Glancing at table 1, we see that all LX instructions have only choice, MLX. So we choose as many MLX templates as their are LX instructions, remove as many M instructions, and then solve for the remaining types. Next, the F types are considered. The B types are considered last, since B instructions only appear at the end portion of bundles.

Table 2 describes DEC, the decrementing function used by the algorithm in table 1. When there are instructions of

more than one type which may fill the same template slot, *i.e.,* when there is a choice between a type and a super type, always choose the instruction from the lowest available type that matches the required functional unit). When there is no appropriate instruction to fill the slot, a NOP is used.

5.3 An Optimal, Recursive Algorithm

The previous section described how to schedule a single instruction group optimally; in this section, we build on this and show how to optimally schedule the entire basic block. Realistically, finding the optimal schedule is impractical, but it is studied because it serves as a good basis for an algorithm that is modified to be feasible. To find the optimal solution, all possible schedules are examined and the best solution found is chosen. We accomplish this exhaustive search by a recursive algorithm, shown in table 3. At each step of the recursion, all possible instruction groups are chosen from among the set of ready instructions. For each choice, a recursive call finds the best schedule of the remaining code.

This algorithm introduces a number of variables to describe various sets of instructions, which we now describe. At any point in the scheduling algorithm, there are some instructions that remain to be scheduled, denoted by the set U. A subset of U, those instructions that are ready to be scheduled (*i.e.,* whose operands are available), is denoted by R. Finally, among R there are again two types: those instructions which must be scheduled this cycle, because their deadlines have been reached, denoted by the set M, and those instructions which may be delayed because they are not yet on the critical path, denoted by the set S. When choosing instructions to schedule in this cycle, only combinations of S are examined, as the members of M must be chosen. Table 3 also introduces the variables C: all instructions chosen from S, P: all instructions chosen for the current instruction group ($P = C \cup M$), and $LeftTypes$: an array that tracks the types of the remaining instructions.

Since table 3 employs an exhaustive search of all possibilities, our methodology is easily extended to provide minimum execution time guarantees, even for real machines. Referring to table 3, when choosing sets of C, we might restrict ourselves to only choices of C which do not produce hazards. In fact, these additional constraints actually serve to reduce the search space, thus allowing our algorithm to run faster. This is an area of future research.

While our current algorithm is only guaranteed to produce minimum cycle code for a non-constrained machine, It is still likely to have fair performance even on an Itanium. This is because code density is proportional to the real performance, since better code density means a smaller I-cache demand, and since better code density effectively increases the machine issue window (by not filling it with NOPs).

5.4 Branch and Bound to Reduce Compile-Time

The above algorithm is too slow, so we present algorithms to speed it up. Some retain optimality, but others do not; this section describes branch and bound techniques that retain optimality. Branch and bound techniques reduce the execution time of an exhaustive algorithm, while retaining the optimality guarantee, by skipping, or *pruning*, those parts of the search space that are known to not contain the best solution. For example, if the current, partial solution has a cost at least as large as the best, complete solution found so far, then regardless of how the remaining instructions are scheduled, they cannot produce a better schedule than the one already found.

From this discussion it is clear that a search order which considers better solutions earlier enables more bounding, thus reducing the search space. Thus, the searching order of the various combinations of S, those instructions that are ready but delayable, greatly affects the compile time. Our philosophy is that a search examining far different solutions early is preferable to one that considers similar solutions first. To, accomplish this we create a bit-vector with each of its bit positions representing one element of S. This vector may have any value in the range from 0 to $2^{size(S)} - 1$. Each value corresponds to a unique choice of S. We use a pseudo-random shift register to change the values of this variable, thereby trying combinations in a pseudo-random order, but also guaranteeing that no solution is tried twice (a property of the pseudo random shift register).

More bounding is possible with a good lower bound on the cost of a basic block or of that portion of a basic block which remains unscheduled. A key insight provides this lower bound:

THEOREM 2 (O(1) COMPUTATION OF LOWER BOUND). *Let U be the set of all remaining-to-be-scheduled instructions. Create an instruction group, U', containing all elements of U, but ignoring the dependencies between the elements. The cost* i.e., *minimum number of NOPs of scheduling U', Cost (U'), serves as a lower bound on the cost of scheduling U, Cost(U).*

PROOF. Consider all possible schedules of U, and choose the best, B. Define another schedule, B' that is identical to B, except that all stop bits are removed. $Cost(B') = Cost(B)$, because each has the same number of templates and the same number of NOPs in those templates. By the property of paired templates, all of the templates of B' are legal. And by theorem 1, B' is a valid schedule of U'. Since the optimal schedule of a group, such as U', is at least as good as any particular schedule, such as U, we arrive at: $Cost(U') \leq Cost(B') = Cost(B) = Cost(U)$. □

Three pruning strategies are presented in table 4. First, $COST\_PRUNING$ uses theorem 2 to find a lower bound on th cost of the current solution, even before it has scheduled all instructions. If the cost of scheduling up to the current point plus the lower bound cost to schedule the remaining instructions is greater than or equal to the best solution found so far, then this path is known to be hopeless and is pruned. The second strategy, $FILLABLE\_NOP$, looks for a template containing a NOP that could have been filled with a delayed instruction, i. It can be shown that the solution where i is included is guaranteed to be at least as good as the solution without it. The third strategy, $SPARSE\_TEMPLATE$, identifies if any selected templates contains a single instruction, where that instruction is delayable. It can be shown that the solution where this instruction is delayed is guaranteed to be at least as good as the current solution, allowing additional pruning.

Where as the methods described in table 4 identify *bad* choices and thus skip them, it is also possible to identifies *good* solutions and quit early (without having to consider the remaining combinations of C. If a given solution is provably optimal, *i.e.,* it has a cost equal to the lower bound,

```
FIND_TEMPLATES(TypeCnt, slot)          // Inputs: # of instructions of each type & starting slot
  enumerate (A, I, M, F, L, B)         // Let A=0, I=1, M=2, F=3, L=4, B=5

  if (slot = 1)                        // To start in slot 1 is necessarily to use template M;MI
    DEC(TypeCnt, M, 1)                 // No choice in this case. 2nd position in M;MI is M
                                       // The decrement function is described in table 2
  if (slot > 0)                        // The only possible templates are M:MI and MI;I
    DEC(TypeCnt, I, 1)                 // In both of these cases, the 3rd position is an I

  DEC(TypeCnt,M,TypeCnt[L])            // Only one template for LX instructions: MLX

// At this point, only A,I,M,F,B remain
  if (TypeCnt[I]≥ TypeCnt[M])          // For Fs, choose between MFI and MMF
    DEC(TypeCnt, I, TypeCnt[F])        // Choose all MFIs
    DEC(TypeCnt, M, TypeCnt[F])
  else
    ExtraM = TypeCnt[M] − TypeCnt[I]
    if (2 * ExtraM ≥ F)                // Check for all MMFs
      DEC(TypeCnt, M, 2*TypeCnt[F])    // Choose all MMFs
    else
      DEC(TypeCnt, M, 2*(ExtraM/2))    // Remove an even number of M types
      TypeCnt[F] −= ExtraM/2           // With MMF, 2 M's for each F

      DEC(TypeCnt, M, TypeCnt[F])      // Use MFIs for those above the limit
      DEC(TypeCnt, I, TypeCnt[F])
      TypeCnt[F] = 0                   // Finished with F
    end else
  end else

// At this point, only A,I,M,B remain
  if (TypeCnt[I] > TypeCnt[M])         // Are there more Is or Ms left?
    Big = I; Sml = M
  else
    Big = M; Sml = I

  gap = (2 * TypeCnt[Sml] − TypeCnt[Big])/3   // Find how many MII-MMI pairs
  DEC(TypeCnt, M, 3*gap)               // Each MMI-MII pair has 3 Ms
  DEC(TypeCnt, I, 3*gap)               // Each MMI-MII pair has 3 Is

// Now, TypeCnt[Big] ≥ 2 * TypeCnt[Sml]
  DEC(TypeCnt, Sml, TypeCnt[Big]/2)    // Fill Big as biased as possible (MMI for M & MII for I)
  TypeCnt[Big]=0

// At this point, only A and B remain
  TypeCnt[A] = 0)                      // MII += NumOfInt[A]/3, since As can be either M or I
  update slot                          // indicate the slot where this instruction group ends

end   // B types naturally go at the end. Trailing NOPs will fill out any remaining slots in the last
      // bundle. The next instruction group is able to overwrite these NOPs
```

Table 1: Optimal Template Selection Algorithm

```
DEC(TypeCnt, Type, Num   // This function decreases the count of the given instruction type by Num.)
                         // A is a super type for either I or M. The strategy is to first decrement the
                         // counter of the more restrictive type, until empty, before removing from A.
  if (Num < TypeCnt[Type])          // See if there are enough of Type
    TypeCnt[Type] −= Num            // Decrease the count for this type
    return
  end if

  Num −= TypeCnt[Type]              // There are not enough of the requested Type,
  TypeCnt[Type] = 0                 // but remove as many as can be done
  if ( (Type = I) or (Type = M) )   // The I and M types can be filled with A types
    if (Num < TypeCnt[A])           // See if there are enough A types
      TypeCnt[A] −= Num             // Decrease the A count
      return
    end if
    Num −= TypeCnt[A]               // Remove all of the A types
    TypeCnt[A] = 0
  end if
  Insert #Num NOPs into remaining templates   // No instruction of the right type found, use NOPs
end
```

Table 2: The DEC function called by FIND_TEMPLATES

```
SCHEDULE_RECURSIVE(U)                  // U = the set of not-yet-scheduled instructions
  ADVANCE_CLOCK(Clk)                   // Advance the clock
  define Best = MAXINT                 // Initially, no best solution
  if (U = ∅)                           // See if finished
    return 0
  define R = READY(U)                  // The set of ready-to-schedule instructions
  define S = NOT_YET_CRITICAL(R,Clk)   // All R which could be delayed
  define M = R − S                     // All R which must be scheduled this cycle
  for each C combination of elements of S   // (including C = ∅)
    P = C + M                          // C+M = this instruction group
    CurTypes = COUNTS(P)               // Counts number of instructions of each type in P
    (Ccost, T) = FIND_TEMPLATES(CurTypes)   // Finds cost of current, and the templates used
    LeftTypes −= CurTypes              // Update the # of each type, still left unscheduled)
    (Ucost, x) = FIND_TEMPLATES(LeftTypes)  // Ignoring dependencies, find minimum cost for U
    if ( not PRUNE(T,Ccost,UCost,S,C))      // Only explore reasonable choices
      Rcost = SCHEDULE_RECURSIVE(U-P)       // Cost of rest (U-P)
    cost = Ccost + Rcost)
    if (Best > cost)                   // Is this solution the best so far?
      Best = cost
    if (Best = Ucost)                  // Is this a minimum-cost solution?
      return Best
    end if
  end for
  return Best                          // All possibiltities have been explored
end
```

Table 3: Optimal Instruction Scheduling Algorithm

```
PRUNE(T, Ccost, Ucost, S, C)                    // Inputs: templates used, current cost, rest cost bound, the set of ready-
                                                // but-delayable instructions, and the set of instructions chosen from S
   define PartialCost = Cost of the partial solution of those already scheduled (everything except P ∪ U)
   define L = S − C                             // The set of instructions that are ready, but not chosen

// COST_PRUNING: rejects solutions which are more expensive than the current best
   if (PartialCost + Ccost + UCost ≥            // Performs true branch and bounding of any path
       BestCompleteSolutionSoFar)               //     known to take longer than a previous solution
          return 1

// FILLABLE_NOP: looks at the chosen templates for a NOP that could have been filled
   if (∃ an instruction, i ∈ L and a NOP, n ⊂ T,    // Checks whether the alternative solution C = C + i
       such that i could have filled the slot of n) //     is better
          return 1

// SPARSE_TEMPLATE: looks for templates that contain only one instruction, where that instruction is able to be delayed
   if (∃ an instruction, i ∈ C and a template, t ∈ T, // Checks whether the alternative solution C = C − i
       such that SCHEDULE_THESE(C-i) = T-t)     //     is better (Doesn't actually call SCHEDULE_THESE)
          return 1
   return 0
end
```

Table 4: Pruning techniques that run in O(1) time

then there is no need to search for a better solution. This optimization is indicated in table 3 by means of the early return from inside the loop.

6. NON-OPTIMAL HEURISTICS TO REDUCE COMPILE-TIME

Non-optimal heuristics are also considered. While branch-and-bound greatly reduces the optimal search space, the compile time for some basic blocks is still unmanageable. In order to reduce the compile time further, non-optimal approaches must be used. Such approaches all operate on the same premise: reduce the compile time by only searching a small portion of the solution space, without guaranteeing that a good solution is found. We use such heuristics only for larger basic blocks, which have already taken a long time to compile, or for those that are longer than a certain threshold. Therefore, most small basic blocks, and some easy-to-schedule larger ones, are scheduled optimally.

In this section, we consider a mixed-bag of such heuristics, the first of which is basic block splitting. Large basic blocks require the longest time to compile, because the number of potential schedules grows exponentially with basic block size. Therefore, it makes sense to break large blocks into smaller pieces that can be optimized separately. Among various partitioning heuristics, we choose the simplest: using the ordering provided originally by SGICC's list scheduler, the basic blocks are split into pieces of the maximum allowed size. In the results section, we evaluate several possible values for the maximum allowed block size, so as to determine the best.

The second heuristic prevents scheduling instructions on certain, non-promising cycles. We only consider those cycles which correspond to the deadlines of any instructions, unless those instructions are schedulable on another selected cycle.

The third heuristic is to limit the for-loop of the base algorithm in table 3. If the statement: "for each C combination of elements of S" is replaced by: "C = all elements of S" then we have an eager scheduling algorithm. If it is replaced by: "$C = \emptyset$" then we have a lazy scheduling algorithm. (All of the instructions are still scheduled, because once their deadlines are reached, the instruction goes into set M instead of S.) Both the eager and the lazy algorithms are very fast, because they only examine one possible solution each. It is more interesting to replace the for statement with: "for each C = either {all of S or $\emptyset$}" In this case, exactly 2 solutions are examined at each level of recursion. Thus the number of searched solutions is in the order of $2^{\#\ of\ allowed\ cycles}$, which is potentially large, but not nearly as large as the original for-loop. We call this approach eager-lazy, because the final solution is an amalgam of eager scheduling for some instruction groups and lazy for others. Our results show that eager-lazy is highly effective: it is able to find much better solutions than either eager or lazy, with only a slightly longer compile time than either.

An alternative to eager-lazy is our fourth heuristic; while eager-lazy examines a binary tree subset of the entire search tree, this method searches the entire tree in a non-optimal manner, by allowing a certain amount of performance loss in the solution. For instance, if the current solution is found to be nearly optimal, then the algorithm might decide that it is good enough and quit. The issue here becomes, what formula to use for the allowed performance loss. In our implementation, the allowed performance loss is based on the slack in the basic block, and the length of time spent in the for loop. The slack of an instruction is defined as the number of allowed cycles on which it could possibly be scheduled and still maintain dependencies and minimum solution cycles; the slack of a basic block is defined as the product of the slacks for the individual instructions. Table 5 details our algorithm for calculating allowed performance loss. This method includes an experimentally determined *multiplying factor* that is discussed in our results.

Our fifth heuristic is a timeout switch. If any basic block, after splitting, takes more than a specified number of seconds in order to compile, then the search simply terminates, and the best solution found thus far is chosen. The selection of the cut-off time is discussed in the results.

Applying the Method Across Basic Blocks Future research may evaluate how this method can be integrated into instruction scheduling across basic blocks. It is possible to modify the algorithm to schedule a trace or a superblock,

```
FIND_ALLOWED_PERFORMANCE_LOSS()        // Finds the number of NOPs tolerated in the solution
   if (iteration# in the current for loop <size(S))   // If the search space has not yet been explored well
      return 0                         // 0 means no performance loss is allowed
   if (size(S < 4)                     // Be optimal for small search spaces
      return 0                         // 0 means no performance loss is allowed
   return itteration# * (log10(Slack)/3 - 1)*   // Slack grows exponentially, so use logarithm.
            MultiplyingFactor)/(size(S))        // Allow more performance loss as the algorithm goes along
end
```

Table 5: Allowed performance loss algorithm

just as the current method schedules a single basic block. In this approach, instructions are moved across basic blocks, in order to improve the code density, and as a by-product, uncovering multi-way branch opportunities. Such an approach does not preclude code motion for other reasons as well.

7. RESULTS

Our algorithm is implemented into the SGICC (v 0.11) research compiler for IA-64. SGICC employs list scheduling with an earliest-deadline-first priority function. Templates are only considered at the last stage, after the order of instructions is already determined. Hence, when the current instruction does not fit into the next slot, a NOP is inserted. Stop bits are inserted whenever the current instruction has a dependency with any instruction already in the instruction group. This is a greedy strategy. SGICC (v 0.11) is run with full optimization. For instruction scheduling across basic blocks, it uses hyperblocking [11], a technique that converts control flow into predication.

Since our algorithm is inserted after the existing instruction scheduling phase of an existing complier, we are occasionally able to take advantage of SGICC solution. Since the list scheduling solution is generated prior to our algorithm, we are able, using theorem 2, to compare the existing solution to our lower bound. If the list-scheduling solution is found to be optimal, our algorithm is skipped. Otherwise, our search algorithms are applied, but the original solution is retained, in case it turns out to yield a better result.

Performance is measured by the number of static NOPs. Static, rather than dynamic NOPs are used, because our algorithm does not currently use consider block frequencies, so the dynamic NOP count would make comparing compile time to performance difficult. Moreover, it is unlikely to be very different. The rational for measuring the results in terms of NOPs is given in section 5.3.

Before running experiments, three parameters from Section 6 must be empirically determined: the size at which to split a basic block, the multiplying factor in determining the amount of allowed performance loss, and the allowed search time. To this end, the equake benchmark was chosen for evaluation. Parameter values are found by the following four steps. First, initial guesses are made for the parameters. Second, one parameter is chosen, and is swept through a range of values, while the other parameters remain fixed. Third, the best value is chosen based on its performance in reducing code size, and its compile-time cost. More emphasis is placed on performance than cost. Fourth, this parameter is updated to the chosen value, and the process is repeated for the next parameter. Table 6 presents the results of the process. It is interesting that only a small percentage of basic blocks time-out, even with a one second cut off. Further, these blocks tend to continue running for a long period of time, if allowed to, without yielding much improvement. Therefore using the cut-off heuristic is worthwhile.

Having determined these parameters, experiments are run on eight SPEC benchmarks, described in table 7. Five of the benchmarks are integer and three are floating point.

Parameter under study	Parameters used in the test: Split Size	Multiply Factor	Allowed Time	Best Value Found
Split Size	*varied*	10^{-3}	60 sec	35
Multiply Factor	35	*varied*	60 sec	10^3
Allowed Time	35	10^3	*varied*	1 sec
Final choices:	35	10^3	1 sec	

Table 6: Choosing Parameters. With initial guesses of 10^{-3} for the multiplying factor, and 60 seconds for the allowed time, the split size is evaluated. A value of 35 is found to yield the best result. Next, the multiplying factor is varied, with the split size now set to 35. The chosen parameter values are shown.

Figure 3(a) shows a comparison of the improvements obtained by three of our methods, as compared to the original SGICC result. Improvement is measured as the percentage of NOPs removed, $1-\#NOPS_{final}/\#NOPS_{original\ SGICC}$. Figure 3(b) shows a comparison of the corresponding compile-time costs. The cost is the percentage increase in compile time, $compile\ time_{original\ SGICC}/compile\ time_{final}-1$. The three schemes are eager, lazy and eager-lazy searching methods. The eager method schedules every instruction that has its dependencies satisfied, while the lazy method does not schedule any instructions that have not reached their deadline. These two approaches do not require any searching for a best solution, since each yields only one solution. Eager-lazy searches for the best among every combination of schedule where the individual instruction groups are either eager or lazy. The x-axes of figures 3(a) and (b) are each divided into three regions: integer results, floating point results and averaged results. From figure 3, it is seen that all of these techniques increase compile time by a modest amount. The performance improvement of even the simpler methods of eager or lazy is 46% for integer and 25% for floating point benchmarks. Eager-lazy must perform better than the other two, since all-eager and all-lazy are two of the many solutions tested by eager-lazy. It gives an improvement of 56% for integer and 35% for floating point benchmarks.

The algorithms to implement the three methods in figure 3 use most of the optimizations outlined in the paper: branch and bound, basic block splitting, skipping unneeded cycles, and timing out. In particular, basic block splitting proved necessary to achieve good compile times for eager-lazy. What is not used in the eager and/or lazy ap-

Name	Description	Type	lines of code	# Basic Blocks*	# Useful Operations*
mcf	Combinatorial optimization / vehicle scheduling	Integer	1909	465	3173
parser	Parses a user's input sequence	Integer	10924	6068	31658
gap	Implements a language and library for group computing	Integer	59481	27651	154362
twolf	placement and global routing package for lithography	Integer	19748	8864	72813
vortex	Single-user object-oriented database transactions	Integer	54550	21764	135335
equake	Simulation of seismic wave propagation	Float	1513	561	5428
lucas	tests the primality of Mersenne numbers	Float	674	279	2375
ammp	Molecular dynamics of a protein-inhibitor complex	Float	13263	4250	38995

* *found by examining the output of SGICC*

Table 7: Benchmarks. All are in C and from the SPEC 2000 suite, with the exception of lucas. The SPEC 2000 version of lucas is in Fortran, so we have used a C version instead.

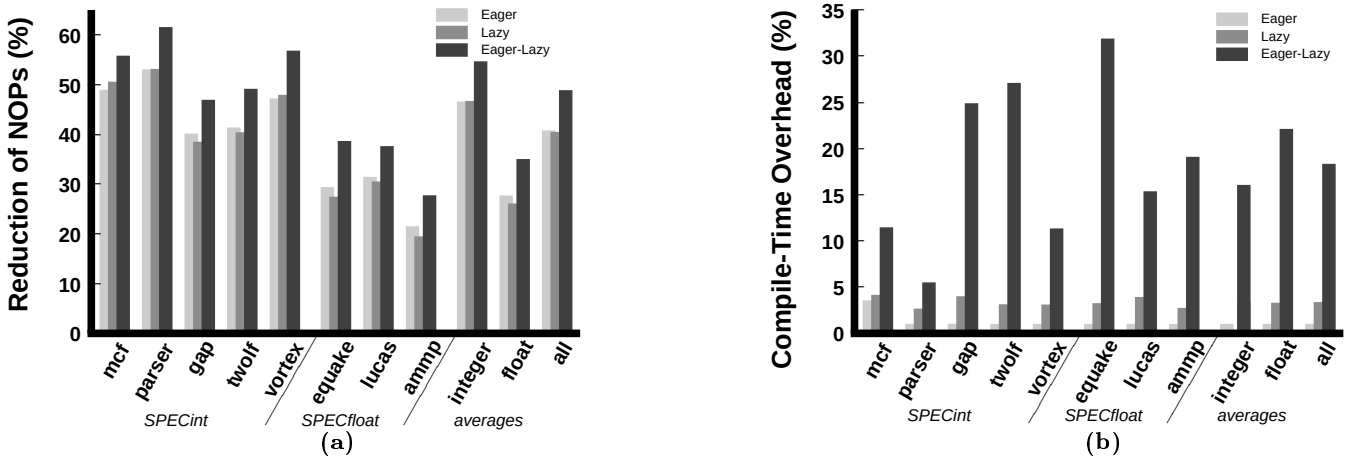


Figure 3: NOP improvement and compilation cost results for eager, lazy and eager-lazy. The first 5 benchmarks are SPEC integer programs, the last 3 are SPEC floating point. Averages are also shown.

proaches are: the allowed-performance-loss technique and two of the pruning methods of Table 4: the *Fillable_NOP* and *Sparse_Template* methods.

Figure 4 shows the performance-loss allowing alternative to the eager-lazy approach, which allows more flexibility in choosing instructions, For this method, the behavior with and without either the *Fillable_NOP* or *Sparse_Template* pruning methods of Table 4. Since the performance for all of these four cases turns out to be so similar as to be almost indistinguishable on a plot, figure 4(a) does not distinguish 4 separate bars. The compile time, however, is markedly different among the cases, especially for the floating point applications. However there is no clear winner among the cases. In particular, equake and ammp show wide differences among the cases, but the preferred cases are opposite for the two benchmarks. The cause of this is unclear.

On examination of figure 4 it may seem odd that the no-pruning technique has the potential to run faster than with pruning; after all, pruning is intended to allow more quickly reaching the solution, and this type of pruning is shown to be optimal. The cause however, is interaction between this optimization and another optimization- the bounding of all solutions known to be worse than the best so far. These prunings are optimal– which means that an optimal solution is guaranteed to remain in the search space after pruning– but this does not indicate that the solution space about to be pruned away does not also contain an optimal solution, since there may be many equally good schedules. In pruning away potentially good, or even optimal solutions, the best-solution-found-so-far variable may not update as quickly as without pruning, leading to a larger search space.

It is interesting that, in both sets of figures, the integer benchmarks result in a greater performance improvement, and therefore also a reduced compile time, due to establishing a better bound on the solution. Table 8 indicates a poorer coded density for floating point programs as well, which is likely due to the limited number of templates available for floating point instructions (see table 1).

Table 8 gives some final numbers for the two most promising techniques. It is seen that code density improvement is similar between floating point and integer cases, but not for compile time and NOP improvement. The NOP improvement and code density improvement measure different things; NOP improvement measures the effectiveness of removing NOPs and the code density improvement measures the relative size of the program, and therefore, roughly approximates the performance.

8. CONCLUSIONS

Constant time algorithms for selecting templates and for finding an upper bound on code size have been described. With these results, an instruction scheduling algorithm that is based on optimal approaches is presented. This method

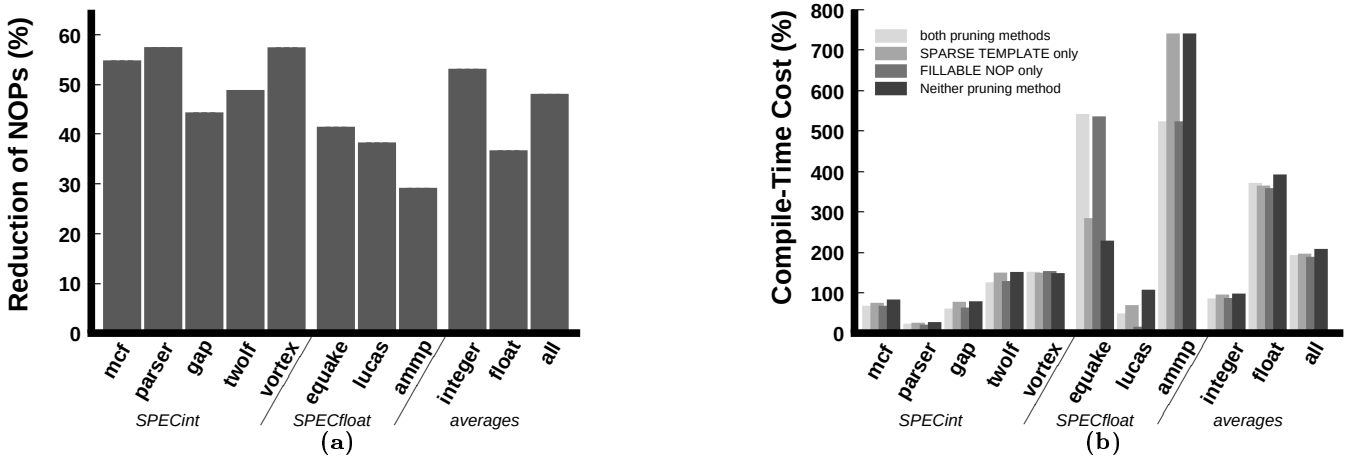


Figure 4: NOP improvement and compilation costs for the allowed-performance-loss heuristic, with different pruning strategies. All of the above use COST_PRUNING, because it is clearly advantageous. It is the SPARSE_TEMPLATE and FILLABLE_NOP methods that are under study. In part (a), the four strategies yielded almost identical results, so the individual bars are not displayed.

Algorithm	average for integer				average for floating point			
	code density	code dens improvemnt	NOP improvemnt	compile time cost	code density	code dens improvemnt	NOP improvemnt	compile time cost
Original	66.87%	0.00%	0.0%	0%	60.23%	0.00%	0.0%	0%
Eager-Lazy	81.33%	14.46%	56.0%	17%	69.85%	9.62%	35.1%	22%
Performance loss	80.83%	14.96%	53.1%	87%	70.41%	10.18%	36.7%	359%

Table 8: Performance of the most promising techniques: Eager-Lazy and Performance-Loss Allowing (using COST_PRUNING and FILLABLE_NOP, as defined in table 4).

is able to remove more than half of the NOPs originally scheduled by the SGICC compiler, for integer programs. Since there are instances where eager-lazy out performs the performance-loss method, and visa versa, a hybrid approach could yield somewhat better results. Modification of the method to include real-machine constraints will also be of interest. These techniques are applied after global instruction scheduling, so that the technique does not replace, but rather supplements, methods such as trace scheduling.

Informal discussions with Intel researchers has identified that low code density is a serious problem, especially impacting the performance of integer programs. This research is a step toward addressing this problem.

9. REFERENCES

[1] J. N. Amaral and G. R. Gao. Using the SGI Pro64 Compiler Infra-Structure for R and D on Back-End Optimizations. In *International Conference on Parallel Architecture and Compilation Techniques (PACT2000)*, Philadelphia, PA, October 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, and K. M. Crozier. Integrated Predicated and Speculative Execution in the IMPACT EPIC Archtecture. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, July 1998.

[3] V. Bala and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, Ann Arbor, Michigan, USA, November 1995. IEEE Computer Society.

[4] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront Scheduling: Path Based Data Representation and Scheduling of Subgraphs. In *Proceedings of the 32nd Annual ACM/IEEE international symposium on microarchitecture on MICRO-32*, pages 262–271. IEEE Computer Society, November 1999.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[6] S. W. Daniel Kastner. ILP-based Instruction Scheduling for IA-64. In *In Proceedings of the ACM SIGPLAN Workshop on Languages*, Snowbird, Utah, USA, June 2001.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, July 1981.

[8] S. Hanono and S. Devadas. Instruction selection, resource allocation and scheduling in the AVIV retargeting code generator. In *Design Automation Conference*, June 1998.

[9] *Intel IA-64 Architecture Software Developer's Manual, Volumes I-IV*. Intel Corporation, January 2000. Also available at http://developer.intel.com.

[10] *Intel(R) Itanium(TM) Processor Hardware Developer's Manual*. Intel Corporation, August 2001.

[11] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann". "effective compiler support for predicated execution using the hyperblock". In *Proceedings of the 25th International Symposium on Microarchitecture*, 1992.

[12] W. mei Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7(1), Jan 1993.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[14] *The SGI Pro64(TM) compiler suite*. SGI Corporation, March 2000. http://oss.sgi.com/projects/Pro64/.

[15] P. Song. Demystifying EPIC and IA-64. *Microprocessor Report*, 12(1):21, January 26 1998.

A Spill Code Reduction Technique for EPIC architectures

– application in the Metrowerks StarCore C compiler –

Virgil PALANCIUC
Motorola DSP Center Romania
313 Spl. Independentei
77206, Bucharest 6, Romania
+40 1 410 54 28
virgil.palanciuc@motorola.com

Dragoş BADEA
Motorola DSP Center Romania
313 Spl. Independentei
77206, Bucharest 6, Romania
+40 1 410 54 28
dragos.badea@motorola.com

Eric FLAMAND
Motorola Metrowerks
29 Bd. des Alpes
38240 MEYLAN France
+33 4 76 61 88 34
eric.flamand@motorola.com

Costel ILAŞ
Motorola DSP Center Romania
313 Spl. Independentei
77206, Bucharest 6, Romania
+40 1 410 54 28
costel.ilas@motorola.com

ABSTRACT.

Graph coloring algorithms have been shown to be an efficient and effective means of performing global register allocation. The power and appeal of these algorithms lies in their strong coloring heuristics and their ability to abstract away seemingly disparate allocation problems such as data-flow constraints, conforming to compiler calling conventions and restrictions due to machine specific details. However, even optimal graph coloring algorithms cannot color every graph. For uncolorable graphs, some live ranges must be spilled to memory to make room for others. The amount of spill code generated and its location can greatly affect a program's performance, so great care should be taken to minimize the amount of spill code inserted. In this article, we present a new approach of reducing spill code, which has the great advantage that it can be used to complement virtually any register allocation algorithm, and provides a good support to implement cheaper spill methods, like spilling to another register (generally, a register from a different class) and rematerialization (reloading the register's value from a constant or expression). The method presented in this paper was partially implemented in the Metrowerks StarCore C compiler where it has proven its efficiency. StarCore is a high performance DSP core that follows an EPIC featured architectural model (VLES – Variable Length Execution Set).

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *compilers, optimization, code generation.*

General Terms

Algorithms, Performance.

Keywords

Compiler, optimization, register allocation, spill code reduction, rematerialization, dataflow analysis, VLES, EPIC.

1. INTRODUCTION

Global register allocation and spilling using graph-coloring techniques has been a topic of practical interest to compiler designers for a number of years. In all compilers that use such a technique, some sort of conflict graph is built whose vertices correspond to the variables and whose edges represent the interference between the live areas of variables. The coloring of the vertices of this graph corresponds to an assignment of the variables to real machine registers. When the number of colors (i.e., real registers) is not sufficient, additional LOAD and STORE statements must be inserted, and these extra statements are referred to as spill code.

Both combinatorial problems, that of deciding whether a graph can be colored with the given number of registers and that of minimizing the amount of spill code, are computationally intractable. This paper presents an heuristic spill reduction algorithm that can be used to complement virtually any register allocation algorithm in order to reduce the amount of generated spill code and thus to produce more efficient (and smaller) compiled code. This algorithm has been developed in the context of the Metrowerks StarCore C Compiler, a highly optimizing compiler that currently targets the Motorola SC140 and SC110 DSP cores. Consequently, the algorithm presented in this paper assumes an architecture with explicit parallelism, in which a set of atomic operations can be executed in parallel. Examples of such architectures are VLIW, VLES, EPIC architectures.

StarCore is based on a variable length execution set (VLES) model; in each cycle, an eight-word instruction set (called *fetch set*) is fetched from memory, and the hardware detects the portion of the fetch set (called *execution set*) that contains the actual set of instructions to be executed in parallel (actually, in the worst case an execution set may span over two fetch sets). Note that the parallel execution of several instructions has to be specified explicitly by the compiler/assembly programmer – i.e. the execution sets are not formed at runtime, instead they are encoded directly in the object file.

There are two main register files in the SC architecture: one is the DALU register file that contains sixteen 40-bit data registers (D-class registers) for integer and fractional data operand storage; the other one is the AGU register file, which contains sixteen 32-bit address registers (R-class registers). Besides these registers, StarCore also has available four address-offset registers and four modulo registers.

StarCore has one predication bit used for conditional execution of instructions, and allows mixed conditional instructions in the same execution set (i.e. we can have in the same execution set instructions that are executed only if the T bit is set and instructions that are executed only if the T bit is not set, or instructions that are executed unconditionally). Regarding the

Conference '00, Month 1-2, 2000, City, State.

control flow change hardware mechanisms of this DSP – although StarCore does not implement branch hints like IA-64 does – it compensates by means of delay slots and hardware loops (the hardware loop mechanism is especially useful for DSP code. It addresses the problem of branching in loops by ensuring zero cycle penalty for all the loop-back changes of flow). For a complete understanding of the StarCore features, you may check the SC140 reference manual [9].

One of the most important decisions the register allocator has to take is which node should be spilled; many different heuristics are used to take this decision, but the idea behind all is the same – the node should not be used frequently, and spilling it to the stack should break as many conflicts as possible. Some register allocators insert some spill code to split the live range of a node into several live ranges, and then spill only one live range of the node. Anyway, once the register allocator has determined which live ranges will be spilled, it must now decide where to place the spill code. The simplest and roughest technique is to insert a store after every definition of the live range and a load before every use. Although this spill-everywhere technique works, it usually generates much more spill code than is necessary. The current register allocations have (usually local) spill elimination heuristics, to improve the quality of the code. Our spill code elimination heuristic is a global heuristic; it can be used for virtually any register allocation algorithm; it has a very good support for rematerialization; and finally – it requires a relatively small implementation effort and it generally produces very good results.

2. PREVIOUS APPROACHES

2.1 Chaitin's Spilling Heuristic

Chaitin's spilling heuristic [5] mentions several refinements to the simple spill-everywhere approach that can reduce the amount of spill code inserted for a given spill decision. These include recomputing live ranges that can be easily recomputed instead of storing them to memory and loading them later and only inserting spill loads at the deaths of other live ranges where register resources are being made available

2.2 Bernstein et al.'s Spill-Almost-Everywhere Heuristic

Bernstein's spill-almost-everywhere heuristic goes further than Chaitin's spilling heuristic in limiting the amount of spill code inserted for a live range in a given basic block. Bernstein's heuristic limits the number of loads and stores inserted into a basic block to one load/store per live range [6]. This in effect renames the live range for each basic block in which it is referenced, whereas Chaitin's heuristic may produce multiple new names in a given block.

2.3 Bergner's Interference Region Spilling

To enable the spilling of limited portions of a live range, Bergner introduces a new concept called the interference region (see [1]). For two interfering live ranges, their interference region is defined to be the portion of the program where they are live simultaneously. By eliminating (spilling) this region from one of the live ranges through the addition of spill code, they will no longer be live simultaneously anywhere in the program, thus they will no longer interfere.

Note that in his PhD thesis, Bergner also presents a new live range splitting technique called interference region splitting, which goes further in the attempt to minimize the spill code.

3. OUR APPROACH

In this article, we present another efficient approach to spill elimination. It was partially implemented on a hierarchical register allocator (which is a refinement of the algorithm presented by Callahan and Koblenz in [3]). We argue that having a separate spill elimination step is generally a good idea, since it provides provisions for a cleaner implementation and one can design a spill elimination algorithm which has the same or better performance as the ones that are currently used in the register allocators (the results shown at the end of this paper support this conclusion). In the remaining part of the article, we will first introduce the basic ideas of the register allocation algorithm we used, and we will present our spill elimination algorithm, plus a few ideas on how to improve the results we obtained in register allocation.

3.1 About Hierarchical Register Allocation

We implemented our spill elimination algorithm on top of a hierarchical register allocator – a slightly modified version of the algorithm presented by Callahan and Koblenz in [3].

Hierarchical register allocation is a graph-coloring algorithm that is sensitive to program flow structure and thus tries to place spill code in less frequently executed portions of the program.

The choice of variables to spill is based on usage patterns between the spills and the reloads rather than usage patterns over the entire program. The method allows a variable to be assigned to one register over a portion of the program, memory in a second portion, and a different register in yet a third portion.

Profiling information can be trivially incorporated to improve the selection of spilled variables and the location of the spill code because all analysis is based on the probability of being in a particular basic block or flowing along a particular control flow edge.

The main idea is to represent the program's loop and conditional structure by a tree of tiles. Tiles are visited in a bottom up fashion and a local interference graph is created and colored (using pseudo registers) for each tile. A tile's local spill decisions are made based on local usage and a compact summary of the local interference graph is passed to the parent tile to be incorporated into its interference graph. After the bottom-up pass has allocated variables to pseudo registers for the entire tree, a top down walk binds pseudo registers to physical registers and introduces spill code where desirable and required, but not necessarily where the decision to spill was made. In addition to better spill code placement, this approach also allows smaller conflict graphs to be constructed.

Although this register allocation algorithm has several important advantages, it has at least one important disadvantage: although it does split the live ranges at tile boundary, it does not split them inside a tile, thus occasionally ending up with very poor code, especially for large loops with high register pressure.

To fix this problem, we had the choice of adding an intra-tile live range splitting algorithm or adding a post-coloring spill reduction step. We did both and we noticed that (even though these algorithms sometimes complement each other very effectively),

the spill reduction step brought far more performance than the intra-tile live range splitting algorithm, basically due to his two main features:

1. It has the ability to correct some of the mistakes made by the register allocation (wrong registers chosen for spilling and for live range splitting)
2. It has a strong support for replacing stack slots with cheaper spill locations (other registers, re-computing the value instead of reloading it etc.)

This observation lead us to the conclusion that the tile-based approach is generally enough for live range splitting, with only one notable exception that we will present later in this paper.

In order to use the spill reduction algorithm after doing hierarchical register allocation (or any other type of register allocation), we had to make one small change in the register allocator: we created a new class of 'special' virtual registers – we called them spill registers. The register allocation algorithm remains unchanged, except that now, when it chooses to spill a virtual register, instead of inserting moves from and to memory, it inserts transfer instructions from and to the associated virtual spill register.

The spill reduction algorithm is activated after the allocator has colored all registers, the code has been changed to physical registers, and the only virtual registers remaining in the flowgraph are the spill registers. The elimination of unnecessary spill code is done in two phases:

1. Gathering data & elimination of spill code.
2. Coloring spill registers.

The first step is actually an additional analysis step that computes equivalence information for all the points (execution sets) in the program (i.e. it detects the registers/immediate values/expressions that have the same value as a virtual spill register); based on this information, we can decide that some load/store instructions are useless because they do not have a significant impact on this equivalence information.

The second step attempts to 'color' the virtual spill registers – i.e. to assign them to different registers. Note that even though we may occasionally have available registers from the same class (due to encoding constraints), generally these register will be registers form a different class and we will be able to use them only as spill locations (and not to perform the actual computation). If no registers are available, we may decide to use the equivalence information provided by the first analysis step in order to re-create the value. Finally, if we have no equivalence information we will end up creating a stack slot and replacing the spill register with that stack slot (actually, if stack consumption is a big issue, we may decide that several spill registers can share the same stack slot).

In the next sections, we will describe in detail the spill reduction algorithm, and we will provide a number of examples to illustrate how it works.

3.2 Phase One

As we already said, in the following sections we are going to view the spill locations as a special class of virtual registers that will be referred as spill registers. We are going to talk about their lifetimes, about their preferences to get a certain color and finally we are going to allocate them real registers. The difference between spill registers and usual virtual registers is that, in case no physical register is available, we are not going to further create other spill locations for them but just associate a stack slot to each spill register. In addition, if the spill registers coloring is not successful, we don't have to further insert spill code (but if it's not successful, we won't be able to eliminate the existing spill code).

The first phase of the spill reduction is based on the computation of a data flow analysis problem – we called it 'equivalence propagation'. We are going to analyze the content equivalence of each spill location with each physical register, with a constant or with a simple expression at any point in program. By "simple expression", we mean only those expressions that can be represented by a single machine instruction. We impose this condition because keeping complex expressions would be impractical in terms of computational complexity; also it will not bring a big benefit, since we are interested in rematerialization of registers at small cost, no larger than that of a reload from a stack slot.

Based on this information provided by the analysis, we will be able to eliminate some unnecessary *load* and *store* instructions right from the first phase and provide enough information to support the second phase of the algorithm in making decisions about coloring or rematerializing the spill registers at smaller costs.

This data analysis step that we are performing is actually a combination of three dataflow problems similar with *constant propagation*, *copy propagation* and a somehow more sophisticated *available expressions* analysis. The difference is that by performing these three analysis steps together, we are able to gather more information as we may be able to make inferences in one analysis based on the information provided by the other (the simplest example is the code sequence r0=3 r1 = 3, where we can decide that r0 = r1 although there was no explicit assignment from r1 to r0.). Actually, the technique we use in this first step can also be considered to be a particular combination analyses and optimisations, and thus it could be described by combining the analysis frameworks, as described in [7].

The lattice we are working on has as elements arrays of sets. For each physical and each spill register we will reserve a position in the array; on this position, we keep a set representing the set of 'objects' that are equivalent with our register (where an object may be another register, a constant value, or an expression). Actually, we do not need to keep a set – we can represent this set as a structure containing:

- the number that is equivalent with the current register (of course, we must be able to assign two special values to that number - 'NOT_A_CONST' and 'ANY_CONST', corresponding to the *bottom* and *top* elements of the "integer constant propagation" (ICP) lattice, as defined in [7])
- a bitset representing the registers that are equivalent with the current register
- a list containing the expressions equivalent with the current registers. As we already said, it is not useful (and it is practically impossible) to have any expression in this equivalence list – we considered that it is enough to keep the 'simple expressions' – i.e. expressions that can be recomputed using a single instruction, and thus

can actually be represented by the instruction itself[*]. Note that here we will also have to keep two special values – ANY_EXPRESSION and NO_EXPRESSION corresponding to the *bottom* and *top* elements of the lattice. We could represent ANY_EXPRESSION by a NULL list, and NO_EXPRESSION by an empty list.

On this lattice, we do a classical iterative data flow analysis (as presented in [7]) with the values initialized to *bottom* (except the values for *entry*, which are initialized at *top*).

```
IterativeAnalysis( FlowGraph FG)
{
  foreach basic_block B in FG do
  {
    if B!=Entry(FG) then
        Initialize(B.equiv_info);
    else
        InitializeEntry(B.equiv_info);
    AddElement(B, WorkList);
  }

  repeat
  {
    Initialize(temp);
    B=ExtractElement(WorkList);
    foreach basic_block P in Predecessors(B) do
      temp = Meet(temp, BlockFlowFunction(P,false));

    if ( temp != B.equiv_info) then
    {
      B.equiv_info = temp;
      foreach basic_block S in Successors(B) do
        AddElement(S,WorkList);
    }
  } until Empty(WorkList);
} /* End of IterativeAnalysis */


EquivInfo Meet( EquivInfo e1, EquivInfo e2)
{
  foreach register R do
  {
    Result.Reg[R].Const = meet_constants(e1.Reg[R].Const,
    e2.Reg[R].Const);

    Result.Reg[r].Regs = BitsetAnd(e1.Reg[R].Regs,
    e2.Reg[R].Regs);
  }
  foreach register R do
    Result.Reg[R].Exp = meet_expressions(e1.Reg[R].Exp,
    e2.Reg[R].Exp);
  return Result;
}
```

[*] One could keep any expression as the tree representation of it, but proving two expressions equivalent would imply very complex symbolic calculations, and it will still have a limited generality. Besides that, we are not interested in very complex expressions since the rematerialization cost would be too high and we will end up preferring the reload from stack, anyway.

```
Initialize(EquivInfo e)
{
    foreach regsiter R do {
        e.Reg[R].Const = ANY_CONST;
        BitsetFill(1,e.Reg[R].Regs);
        e.Reg[R].Exp= ANY_EXPRESSION;
    }
}

InitializeEntry(EquivInfo e)
{
    foreach regsiter R do {
        e.Reg[R].Const = NOT_A_CONST;
        BitsetFill(0,e.Reg[R].Regs);
        e.Reg[R].Exp= NO_EXPRESSION;
    }
}
```

Notice that during the analysis we call the block flow function with a second parameter 'false'. This is because we use the same function (with minimal modifications) in the elimination of the useless spill instructions.

The meet operator we used relies on the classical meet operators for ICP and for bit vectors. In the case of expressions, lattice and the meet operator is somehow similar with the one used for constant propagation – here we have $e1 \wedge e2 == e1$ if e1 and e2 produce the same result (assuming neither e1 or e2 have the special values of *top* or *bottom*).

The equivalence propagation analysis mostly provides us information about which load instructions are to be removed and then we can decide to remove store instructions that are useless (they actually become dead code). This happens because usually load instructions can exist in equivalence points, where $r_j == s_k$. Notice that stores usually come either after definitions of physical registers (which invalidate possible equivalence relations existent up to that point) or after redefinitions that update the old value. Thus, it is unlikely to directly discover useless store instructions – but here it is an example (figure 1) in which both the redefinition (mac) and the store instruction can be removed.

```
add r1,r2,r3
store r3,s1
[...]
load s1,r4
mac r5,r6,r4 /* equiv_info:{(r5==0)}*/
store r4,s1
[...]
```

Figure 1. Both redefinition and store can be removed

Anyway, this is an exception – store instructions are usually seen as necessary in the first place and they may eventually become useless after removing some reloads (if we manage to remove all the loads exposed to a store).

Consider the example in figure 2, in which variable *x* is spilled at a certain point in register allocator. The temporaries *t1, t2* and *t3* are created and let's assume that real registers can be assigned to them. If *t2* and *t3* get the same physical register (*r2*) and there is no other rewrite instruction on *r2* between instructions *i3* and *i6*,

then the values in s1 and r2 are identical before instruction *i6*. As result, we can optimize this piece of code and eliminate the load in *i5*.

def x *use x* *use x*	
def t1 *store t1, s1* *load s1, t2* *use t2* *load s1, t3* *use t3*	*i1. def r1* *i2. store r1, s1* *i3. load s1, r2* *i4. use r2 /*no redef of r2*/* *i5. load s1, r2 /* unnecessary load */* *i6. use r2*

Figure 2. Code sequence containing a useless load

In figure 3, we have an example in which store instructions can be eliminated. Assume variable *x* is dead at block exit. If *t3* and *t4* were given the same color (*r3*), the data flow analysis will see $s_1==r_3$ at point *i8* in program. The algorithm goes on with removal of the load *i8*, which used to be the last instruction using the spill register *s1*. Spill registers are analyzed as usual virtual registers and *s1* is going to be seen as *dead* after program point *i7*. This means that its definition in *i7* is dead code as well and can be removed.

def x *use x* *use x & def x* *use x /*last use of x*/*	
def t1 *store t1, s1* *load s1, t2* *use t2* *load s1, t3* *use t3 & write t3* *store t3, s1* *load s1, t4 /*last use of s1*/* *use t4*	*i1. def r1* *i2. store r1, s1* *i3. load s1, r2* *i4. use r2* *i5. load s1,r3* *i6. use r3 & def r3* *i7. store r3,s1 /* dead code after removing i8 */* *.....no def of r3* *i8. load s1, r3 /* last use of s1*/* *i9. use r3*

Figure 3. Dead code store (instruction i7) due to load elimination (instruction i8)

Next, we present the remaining routines from the first stage - *BlockFlowFunction* (which is the 'core' of the analysis algorithm) and *FirstPhase*, which is the entry point in this phase:

```
void FirstPhase(FlowGraph fg){
   IterativeAnalysis(FG);
   foreach basic_block B in FG do
      BlockFlowFunction(B,true);
   DeadCodeElimination(FG);
}
```

```
EquivalenceInfo BlockFlowFunction(Block b, bool elim)
pp_equiv_info = b.equiv_info
foreach ES in b do
   foreach instruction I in ES (traversed so that data dependencies
   are fulfilled) do
   {
      if (I is a call instruction) then
         foreach register r destroyed by callee do
          delete_equiv(pp_equiv_info, r);
    if(I does not change any register) then continue;
      if(I reloads register r from memory ) then
         delete_equiv(pp_equiv_info, r);
      if(elim && redundant(pp_equiv_info, I)) then
         remove_inst(I)

      if(!redundant(pp_equiv_info,I)) then
      {
         delete_equiv(pp_equiv_info, dst(I));
         add_equiv (pp_equiv_info,I,dst(I));
      }
   }
   return pp_equiv_info;
}
```

The heart of the analysis algorithm is the function *BlockFlowFunction*, which produces the EquivalenceInfo at the end of a basic block, given the EquivalenceInfo at the beginning of the basic block. In this paper, we have presented a simplified version of it, that does not deal with issues such as predicated execution or unpairable execution sets (i.e execution sets with circular data dependences – for example the 'swap' execution set: tfr d0,d1 & tfr d1,d0). The analysis over one basic block is done as follows: for each instruction that changes at least one register* we first check to see whether the instruction is redundant (i.e. writes a value that was already existing in the register) and if it is, then we can safely remove it (if we are eliminating useless instructions – i.e. if the value of second parameter is 'true'). If it is not redundant, we delete all equivalences of that register, and re-create a new set of equivalences. We do this by using three functions: *delete_equiv*, *add_equiv* and *redundant*. While the effect of the first one is obvious, the other two need some additional explanations.

The function *redundant* checks to see whether the computation of an instruction was not already performed. It can be easily implemented using the *add_equiv* function – it simply checks whether this instruction writes any register; if it does, then it is redundant if it would bring no new equivalence information for that register.

The function *delete_equiv* removes the equivalence information related to a register – i.e. it initializes the information for that register to *bottom* and removes all other equivalence information related to it (i.e. if we destroy r1, we will need to destroy the equivalences r2=r1 and r3=r1+1).

The function *add_equiv* is the most complex function used in this stage. It contains an iterative inference engine that first extracts the obvious equivalence information provided by the instruction

* Here by register we also mean spill register; Also note that call instructions are considered to destroy all physical registers that are not callee saved (according to ABI), but no spill registers

(i.e. if we have tfr d1,d2, we know that after this instruction d1 will be equal to d2) and then updates all equivalence information where the register is involved, considering the existing equivalence information. For example, in the case of a tfr instruction from d1 to d2, the first step is to mark the fact that d1 is equal to d2 and d2 is equal to d1; but after that, we also need to extend the existing equivalence information (i.e. if we knew that d1 is constant and has the value '3', we must mark the fact that d2 equals 3; further, if we knew that d3 is d1+d6, we must add the equalities d3=d2+d6, d2=d3-d6, d6=d3-d2). Note that this function has to also take into account the arithmetic properties of an operation (e.g. commutability).

Note that, besides removing redundant instructions, we could also replace spill registers with expressions, thus providing rematerialization. However, doing this from the first phase may lead to wrong rematerialization decisions which could adversely impact on the second stage, so it is a good idea to delay rematerialization until the second phase.

3.3 Phase Two

In the second phase, spill registers are bound to physical registers, constants/expressions (when the value may be re-materialized, and this has not been done in the previous step) or memory locations, according to the designated preferences & priorities.

```
SecondPhase(FlowGraph FG, EquivalenceInfo EI){
   Webs = ComputeWebs(FG);
   foreach web W in Webs do
      RematerializeAll(W, EI, false); //(1)
   IGraph = ComputeInterferenceGraph(Webs);
   Pri = ComputePriorities(Webs, IGraph);

   foreach web W from Webs, sorted by Pri do
   {
      RematerializeAll(W, EI, true); // (2)
      if(no remaining uses for W) then continue;
      Color = ChooseBestColor(W,IGraph, EI);
      foreach rematerialized use from W do
         if (reload from Color is better than current
         rematerialization instruction) then
            Change the rematerialization instruction to a reload
            instruction;
      Adjust EI to reflect destroyed equivalences;
   }
   DeadCodeElimination(FG);
}
```

Note that in the coloring stage, we do not color virtual spill registers, but instead we color webs (roughly speaking, a web is a distinct lifetime of a register - for a more precise definition see [7]). This is because a spill register is likely to have several lifetimes (and this likelihood is further increased by the first spill elimination stage). Consider the example in figure 4.

```
def x
store x,s0
.....
load s0,x
[ a large portion of the program that uses x ]
store x,s0
.....
load s0,x
```

Figure 4. Spill register with two lifetimes

In this example, the spilled variable creates a spill register with two distinct lifetimes, which can (and actually should) be colored individually. This case is quite frequent in hierarchical register allocation, where the 'large portion of the program' may be an inner loop where x was allocated to a physical register, and outside the loop x was spilled. In this case, we would have fixup loads&stores that would follow exactly the pattern presented in this example.

Going back to the coloring algorithm - notice that immediately after building the webs, we make a first call to *RematerializeAll*. This procedure uses the equivalence information to replace all possible reloads with equivalent rematerialization instructions (constants, transfers from an equivalent register, recomputation). However, we don't want all the values to be rematerialized from the very beginning – otherwise we would have done this from the first stage. Instead, at the first call to *RematerializeAll* (denoted with (1) in the pseudocode) we only rematerialize the values that do not stretch any lifetimes (we don't do rematerialization if it would bring additional conflicts between physical and spill registers). Consider the example in figure 5.

```
def  r1
def r2
add  r1,r2,r3
store r3, s0
def r3 // destroys the old value from r3
[...region with high R-class register pressure...]
[...region with high D-class register pressure...]
(*)load  s0, r3
```

Figure 5. Rematerialization would stretch lifetimes of r1 and r2

In this example, we have a region with high R register pressure, which makes us spill r3 and re-use it for different purposes. Assuming that we have another region with high D register pressure, we may not want to re-materialize r3 in the point (*), because doing so would stretch the lifetimes of r1 and r2 and thus would prevent these two registers from being used as spill locations for the D-class registers in the region where we have a high D-class register pressure. It is better to delay rematerialization decisions that stretch lifetimes until the point (2) in the algorithm, where we have to assign a color for the spill register (and at that point we know that this spill register is the

most important one which was not already colored, thus rematerialization is acceptable even if it stretches lifetimes).

After the first rematerialization step, we build the interference graph and we compute the priority of each spill register (actually, for each web). The register with a higher priority will be the first to be colored, as it has a higher impact on program speed. The priority is computed with a heuristic merit function similar to a function that computes spill cost during register allocation.

Having the interference graph built and the priority computed, we start the actual coloring. First, we attempt to rematerialize as many uses as possible from the current web. If there is no register-use left to be colored, our job is done and we may continue with the next web. Otherwise, we re-evaluate the conflicts (if we rematerialized some uses, we may have a smaller web with less conflicts) and we select the best color available. A good heuristic for the 'best' color is that color which could be used by the fewest neighbors (we are only interested in the neighbors that are spill registers) and which destroys as few rematerialization opportunities as possible. For example, if we have a physical register Rx that is in conflict with all the neighbors and it cannot be used to rematerialize any neighbor, then Rx is likely to be the perfect choice for a spill location. Note that we may also have special preferences for some registers (for example – if we somehow manage to get a register from the same class, we will have smaller register-to-register transfer costs).

After having chosen a register for the spill location (or a stack slot, if no register was available) we make an additional check to see whether it would not be profitable to replace some of the rematerialized instructions with reloads without loosing performance and without creating additional conflicts. Replacing rematerialization instructions with a register that was chosen as spill location is useful because it is usually cheaper – not only for speed, but also for code size. For example – we may have no speed penalty when materializing register from a constant, but it is likely that we will have a code size penalty. In addition, reloading from the register that was chosen as spill location could avoid stretching the lifetime of several other physical registers.

Assuming that we did pick a register as spill location, we need to take care of the possible inconsistency that may occur in the equivalence information provided by the first step. For example, in the case presented in figure 5, if we chose r1 as spill location for a D register in the region of high D-class register pressure, we must know that we will not be able to rematerialize r3 anymore in point (*).

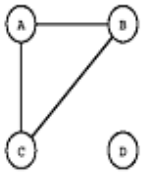
Finally, in the end of the second stage of the algorithm we have to call dead code elimination again, since generally this coloring stage leaves a lot of dead code behind.

3.4 Example

We will now examine a somehow more elaborate example that proves the power of this spill elimination algorithm (this is in fact exactly the example presented by Bergner in [1]).

Assume that we have the code in figure 6 (with the corresponding spill costs) and that there are only two available registers left. It is obvious that this graph cannot be colored, and we will have to spill the variable A.

```
A=input();
B=A+1;
if(A) {
   C = A+2;
   B = A+C;
   if(C) {
      B = B+C;
      C = B+C;
   }
   A = B+C;
}
D = A+B;
```



node	cost
A	8
B	12
C	12
D	∞

Figure 6. Code example, with corresponding interference graph and spill costs

A regular register allocator, using Chaitin's local heuristics for spill reduction, would generate the code presented in figure 7(a), which has two store instructions and two load instructions. As described in [1], the interference region spilling heuristic would produce the code shown in figure 7(b) that has only one load instruction and one store instruction. Our method would give the optimal result that is shown in figure 7(c) (which would actually be the code generated by using interference region splitting; but remember that our method is not a live range splitting algorithm, it is a spill reduction algorithm and thus can complement any register allocation algorithm already implemented).

```
r0=input();
Store r0;
r1=r0+1;
if (r0){
  Load r1;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;
  Store r0;
}
Load r0;
r0=r0+r1;
```

```
r0=input();
Store r0;
r1=r0+1;
if (r0){
  Load r1;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;

}

r0=r0+r1;
```

```
r0=input();

r1=r0+1;
if (r0){
  r1=r0;
  r0=r1+2;
  r1=r1+r0;
  if (r0){
    r1=r1+r0;
    r0=r1+r0;
  }
  r0=r1+r0;

}

r0=r0+r1;
```

Figure 7. Results of spill reduction using: **a**)Chaitin's spilling heuristic; **b**)Bergner's IR spilling heuristic; **c**)Our approach

Let's explain on short why our method produces the result in figure 7(c): after doing the first spill reduction phase, it is obvious that (on this example) we end up with a code very similar with the one produced by IR spilling (the last load would be eliminated because the value of A already exists in r0; the last store will be eliminated as it is dead code).

But as opposed to other spill reduction approaches, instead of spilling directly to the stack, we are now trying to 'color' the virtual spill register (eventually, if we find no color or no way to re-materialize its value, an available stack slot will be assigned to this spill register). Looking at the code, we see that at the point

where we have the first load instruction, we have the value already available in r0, so we can 'reload' the value directly from r0. Thus, the first store will become redundant and be eliminated as dead code.

Note that in this example that we are actually facing a special case of the problem of rematerialization: we could have decided from the very first step that it is possible to remove the first load instruction and replace it with a register-to-register transfer. In this particular case, the results would be the same, but in general, taking this decision in the first step would stretch the lifetime of r0 before the 'coloring' step. In a different example, we may have preferred to reload A from stack, and to use the r0 register as a spill location for another (more important) variable.

3.5 Support for the algorithm

In this section, we will talk about the improvements that can be made (generally in other optimization steps) to achieve better results with this spill elimination algorithm.

First – it is quite obvious that it would be best if the spill elimination algorithm had nothing to do – which means that in order to achieve good register allocation, it is very important to have a lifetime-sensitive software pipelining algorithm (good examples of such algorithms are Slack Modulo Scheduling and Swing Modulo Scheduling – but depending on the architecture, other algorithms may prove to be better). In addition, if the register allocator follows an instruction-scheduling step – it is best to have a bi-directional, lifetime-sensitive instruction scheduler as the first instruction-scheduling step.

Another very important change that should be made in order to achieve good register allocation is restriction graph splitting, which is actually a kind of 'architecture-dependent live range splitting'. By this, we mean that if the architecture has encoding constraints that force several registers to be colored in a single step, the register allocator should build a restriction graph and should check from that graph whether all registers are colorable. If a register is determined to be uncolorable from the very beginning, we should split its live range to avoid spilling it. For example, assume the following code sequence*:

```
[
 max  vd0, vd1
 max vd0, vd2
]
```

In this case, even assuming we have plenty of physical registers available, we cannot assign physical registers to vd0,vd1 and vd2 (if we assign d0 to vd0 then both vd1 and vd2 must be d4 – but vd1 and vd2 cannot share the same register – first because we would have two writes to a register in the same cycle, and even assuming we would ungroup the instructions – if neither of them is dead code, then vd1 and vd2 have a conflict in the conflict graph.). Normally, we would have to spill one of the registers – but a better solution is to split the live range of one register as follows, in order to obtain two colorable restriction graphs instead of an uncolorable one:

```
tfr vd0, temp
[
 max  vd0, vd1
 max temp, vd2
]
```

Note that efficiently splitting the conflict graphs in a way that enables the registers to be colored, and by adding a minimal amount of transfer instructions, is a problem by itself (and not at all a trivial one). However, we do not discuss this problem here.

If the register allocator has a mechanism for preferences, it is useful to add preferences in such a way that all temporary registers that result after spilling a virtual register will prefer the same color - thus enabling the spill elimination algorithm to remove the useless load/store instructions without leaving register-to-register transfers behind. Note that even if the register allocation algorithm has no mechanism for preferences, we can still avoid useless transfers (but a little bit less efficient) using the following peephole optimizations:

- This peephole optimization increases parallelism and provides more opportunities for the following peephole optimizations

$$\left.\begin{matrix}\text{tfr } R_x, R_y \\ \text{use } R_y\end{matrix}\right\} \Rightarrow \left\{\begin{matrix}\text{tfr } R_x, R_y \\ \text{use } R_x\end{matrix}\right.$$

- These two peepholes are designed to remove the useless transfer instructions:

$$\left.\begin{matrix}\text{tfr } R_x, R_y \\ [\text{no\_use / redefinition of } R_y]\end{matrix}\right\} \Rightarrow \text{remove tfr } R_x, R_y$$

$$\text{tfr } R_x, R_x \Rightarrow \text{remove tfr } R_x, R_x$$

- Finally, we may add the following (somehow more sophisticated) peephole that removes transfer instructions, at the same time increasing parallelism:

$$\left.\begin{matrix}[\text{definition of } R_x] \\ \text{tfr } R_x, R_y \\ [\text{definition of } R_x]\end{matrix}\right\} \Rightarrow \left\{\begin{matrix}[\text{definition of } R_y] \\ \text{remove tfr } R_x, R_y \\ [\text{definition of } R_x]\end{matrix}\right.$$

4. RESULTS

In this section, we present a set of results that prove the efficiency of our spill reduction algorithm. These tests consist mainly of optimized and un-optimized DSP code (which is our primary interest). However, good results have been reported for control code, too.

* This example is SC100 assembly code. On StarCore, there is a restriction which forces a distance of +4 between the indexes of the two registers that serve as operands to a *max* instruction; also both operands must be from the same bank – i.e. it is disallowed to have one operand in the interval d0-d7 and the other one in d8-d15.

We tested this optimization in the context of the Metrowerks C Compiler for StarCore, on a set of large applications as well as on several DSP benchmarks. The applications are:

- G729 – MDCR[1] optimized C implementation for the ITU-T G.729 8 kbps speech vocoder
- G729a – MDCR optimized C implementation for the ITU-T G.729 Annex A 8 kbps speech vocoder
- EFR – Out-of-the-box implementation of the GSM Enhanced Full Rate 12.2 kbps speech vocoder
- AMR - Out-of-the-box implementation of the 4750 ... 12200 bits/s speech codec for Adaptive Multi-Rate speech traffic channels
- G723 – Out-of the box implementation of the ITU-T G.723 dual-rate speech codec.

4.1 Results for the optimized G729 vocoder:

	encoder average	***encoder worst-case***	***decoder average***	***decoder worst-case***
With spill reduction	**90752**	**95171**	**15922**	**17360**
Without spill reduction	**101168**	**106270**	**16528**	**18035**

Average improvement: 3.8% decoder, 11.5% encoder

4.2 Results for the optimized G729a vocoder:

	encoder average	***encoder worst-case***	***decoder average***	***decoder worst-case***
With spill reduction	**51863**	**52766**	**9932**	**9937**
Without spill reduction	**52449**	**53432**	**10214**	**10220**

Average improvement: 2.8% decoder, 1.1% encoder

4.3 Results for EFR:

With spill reduction	**Without spill reduction**
354440 cycles	360085 cycles

Improvement: 1.6% (single frame test, coder+decoder)

4.4 Results for AMR:

	encoder average	***encoder worst-case***	***decoder average***	***decoder worst-case***
With spill reduction	**164628**	**288883**	**37997**	**47733**
Without spill reduction	**172769**	**296472**	**38250**	**48180**

Average improvement: 0.66% decoder, 4.95% encoder

[1] MDCR stands for Motorola DSP Center Romania

4.5 Results for G723:

	encoder	***decoder***
With spill reduction	**732487**	**73123**
Without spill reduction	**733720**	**73213**

Improvement: 0.12% decoder, 0.16% encoder

4.6 Results for un-optimized DSP benchmarks:

File name	**Cycles (with spill reduction)**	**Cycles (without spill reduction)**	**Impro-vement**
cor_h.c	1505	1530	1.6%
levinson.c	1550	1603	3.4%
mb01.c	17839	17839	0%
mb02.c	2660	2660	0%
mb03.c	29023	31490	8.5%
norm_corr.c	11488	11488	0%
pitch_fr3.c	996	1040	4.4%
qua_gain.c	1820	2016	10.7%
search_10i40.c	10391	11273	8.5%
vq_subvec.c	850	1133	33.3%

Notes:

- The results are in cycles on the Motorola StarCore®140 DSP core.
- The performance increase is more significant on the encoders compared with the decoders. This shows that the encoders are more complex than the decoders, and have significantly higher register pressure.
- Where we have 0% improvement, it is due to good register allocation (no spill) – no improvements could be done to it.

5. CONCLUSIONS AND FUTURE WORK

While graph coloring is widely recognized as 'the method' for solving register allocation problems, the problem of inserting spill code has not seen yet a solution with such a wide acceptance. Several heuristics have been developed in order to tackle this problem, and there are generally two directions in attempting spill code reduction:

- live range splitting register allocators
- spill reduction heuristics

Aggressive register allocators use both these approaches to try to minimize the spill code – and we have also taken this approach. We used a hierarchical register allocation algorithm that splits the live ranges and inserts spill code in cheaper points (considering the program's control structure) combined with a Chaitin-style spill reduction heuristic. However, this algorithm produced unsatisfactory results when dealing with very large tiles that had high register pressure. We had the choice of adding an intra-tile live range splitting algorithm or designing a better spill reduction algorithm. In our first attempt, we tried to implement a live-range splitting algorithm similar to the one described by Chow (see [4]) in the existing register allocation algorithm, which proved to be a

tough task since the two algorithms use radically different approaches. We ended up implementing a live range splitting algorithm that was executed before the actual coloring of the tile – actually, it was only a code transformation designed to reduce register pressure. We implemented two different live range splitting heuristics:

- the first heuristic starts by choosing 'split points' as being points where we have a relevant increase or decrease in register pressure, then at each split point it splits those variables that have high/low spill costs and a high usage pattern both before and after the split point
- the second approach starts by selecting the live ranges to split (those with a low/high spill cost) ant then decides how many times and where to split the live range.

The results yielded by these algorithms were disappointing. Although we had good (sometimes excellent) results on some test cases, on other test cases we had significant performance decrease (going as high as 15-20%).

Note that our live range splitting heuristics do not fit well in a Briggs-like register allocator (since they start from the ideas presented by Chow and Hennessy in [4], which have a different coloring framework). Bergner's interference region splitting algorithm presented in [1] is a more natural complementary live range splitting solution for Briggs-like register allocators (as well as for our hierarchical register allocation algorithm); Still, we did not choose to implement it as the results presented in his PhD thesis showed performance decrease similar to that observed for our live-range splitting approach.

The results of our efforts towards implementing a complementary live range splitting algorithm (as well as the results presented by Bergner) led us to the conclusion that (except for the particular cases of architecture-specific restriction graph splitting algorithms described in section 3.5) it is generally sufficient to do live range splitting at tile boundary. In order to further improve the results of hierarchical register allocation, we need to have a good spill reduction heuristic – and the results obtained using the solution presented here make us believe that this algorithm is a fairly good solution for spill reduction. The great advantage of our approach is that (as opposed to Bergner's IR spilling/splitting) this approach always yields better (or equal) performance when compared with that provided by the supporting register allocator alone (in this paper, - a spill reduction heuristic similar in spirit with the one presented by Chaitin).

As future purposes, we plan to fully implement the spill reduction algorithm in our hierarchical register allocator and to evaluate the benefits of the full implementation. In addition, we want to experiment with new heuristics for rematerialization and coloring. From the implementation's point of view, we want to design a very efficient algorithm for the iterative engine that updates the equivalences, based on a more efficient representation for expressions (and maybe try something similar with global value numbering). It should also prove interesting to know what is the performance of the spill reduction algorithm when used with a non-hierarchical register allocator, and what other improvements it needs for becoming truly efficient when combined with other register allocation algorithms.

Last, but not least – the algorithm requires a more detailed evaluation, on smaller test cases (but with high register pressure) - we have yet to discover what it does best and where it can be improved.

6. REFERENCES

[1] Peter E. Bergner. Spill Code Minimization Techniques for Graph Coloring Register Allocators. Ph.D. Thesis, University of Minnesota, 1997

[2] Preston Briggs. Register allocation via graph coloring. Ph.D. Thesis Rice COMP TR92-183, Department of Computer Science, Rice University, 1992.

[3] David Callahan, Brian Koblenz. Register Allocation via Hierarchical Graph Coloring. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28, 1991.

[4] F.W. Chow and J.L.Hennesy. The Priority-Based Coloring Approach to Register Allocation.

[5] Gregory J. Chaitin. Register allocation and spilling via graph coloring. SIGPLAN Notices, 17(6):98-105, June 1982. Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.

[6] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. SIGPLAN Notices, 24(7):258–263, July 1989. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.

[7] Cliff Click and Keith D. Cooper. Combining Analyses, Combining Optimizations. ACM Transactions on Programming Languages and Systems. Vol. 17 No. 2., March 1995, Pages 181-196

[8] S. Muchnick. Advanced Compiler Design & Implementation. Morgan Kauufmann Publishers

[9] SC140 DSP Core Reference Manual. http://e-www.motorola.com/brdata/PDFDB/docs/MSC140DSPCORERM.pdf

Incorporating Predicate Information Into Branch Predictors

Beth Simon Brad Calder Jeanne Ferrante

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{esimon,calder,ferrante}@cs.ucsd.edu

1. INTRODUCTION

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as a viable architecture for achieving the *instruction level parallelism* (ILP) needed to keep increasing future processor performance [8]. The IA-64 Itanium processor [1] is an example of an EPIC architecture. An EPIC architecture issues wide instructions, similar to a VLIW architecture, where each instruction contains many operations.

One of the new features of the EPIC architecture is support for *predicated execution* [14], where each operation is guarded by one of the predicate registers available in the architecture. An operation is committed only if the value of its guarding predicate is true.

One advantage of predicated execution comes from predication's ability to combine several smaller basic blocks into one larger region. This provides a larger pool from which to draw instruction level parallelism (ILP) for EPIC architectures. Another advantage of predicated execution is that it can eliminate hard-to-predict branches by translating them into predicate defines, which do not need to be predicted. This comes at the cost of executing both paths following the branch as if it were a single path.

Choi et al. [7] recently performed a study, where they reported that only 7% of cycles are spent due to branch mispredictions for the SPEC 2000 integer benchmarks (without using if-conversion). This is partially due to the in-order Itanium processor stalling because of memory latencies, which end up shadowing the stalls due to branch mispredictions. As this type of EPIC architecture progresses and memory latencies are better hidden, the stalls due to branch mispredictions will have a much larger impact.

The goal of our research is to use predicated execution to see how low we can make the branch misprediction rate by removing all of the the hard-to-predict branches, not caring about the increase in executed code created via predication. We first examine how to create predicated regions to remove hard-to-predict branches. In order to aggressively form predicated regions around these hard-to-predict branches, we had to leave unbiased, but originally predictable, branches (conditionals, unconditionals, and returns) inside predicated regions. We call branches left inside predicated regions *region branches*.

The creation of predicated sequences (region formation) based on removing a hard-to-predict branch can have a negative impact on the predictability of these region branches. Region branch execution is now predicated on a register defined by a predicate compare definition that was added in order to remove the hard-to-predict branch. These region branches will need to be predicted during fetch more frequently than they were in the original, non-predicated code (i.e. a region branch will be fetched both when its guarding predicate will be true and when it will be false). This can cause what we call *misprediction migration*, where the poorly predictable pattern of a hard-to-predict branch that was eliminated due to predication is merely migrated to a region branch. In addition, direct branches (e.g., unconditionals and returns) that are left in the region are also affected by misprediction migration. Before the region was formed, these region branches were accurately predictable as taken. After region formation, they now need to be predicted as either taken or not-taken when guarding predicate is TRUE, and should always be predicated as not-taken when the guarding predicate is FALSE. Because of misprediction migration, we found little improvement in branch misprediction rate for some programs when using a traditional branch predictor for region formation targeted at hard-to-predict branches.

In this paper, we examine two new branch predictor optimizations. First is a new branch prediction optimization called *Squash False Path* (Squash-FP) that attempts to know a branch's guarding predicate value as it is being fetched, and, if it is false, then the branch is predicted as not-taken. The goal of this predictor is to correctly predict the region branches that are on the false path as not-taken.

The second predictor we examine adds predicate information into the global history register. We examine the *Predicate Global Update Branch Predictor* (PGU) architecture that incorporates predicate information into the global history to try and improve the performance of region-branches that benefit from correlation. The PGU predictor updates the global history with the predicate result when the predicate defining instruction is resolved. This can allow region branches to benefit from this history correlation when making their prediction from the global prediction table.

Region branches only benefit from these two branch prediction architectures if they are scheduled far enough apart from their predicate definitions. Therefore, we examine the benefit of rescheduling the predicated region to move the region-based branches as far away as possible from their predicate defining instructions. This attempts to increase the cases where a predicate define can be resolved before the branch is fetched, so it can be used to form the prediction for the branch.

2. PRIOR WORK

2.1 Branches and Region Formation

Previous region formation techniques have focused on using predicated execution to group basic blocks from various control flow paths into one region to improve compiler optimization opportunities and scheduling [12] [4] [3]. These hyperblocks are typically formed from an inner-most loop body. Basic blocks are incorporated into a region based on a heuristic function that weighs the block's frequency of execution and size in terms of instructions in relation to the main path of the hyperblock being formed. Hyperblocks target unbiased branches by translating them into predicate defines and incorporating the subsequent paths in the predicated region. Hyperblocks only allow heavily biased branches to remain as predicated branches in the region and incorporate the frequent path of execution as part of the region. Our region formation methods will target the removal of only unpredictable branches rather than unbiased branches.

2.2 Interaction Between Predication and Branch Prediction

Mahlke et al. [11] investigated the interaction of predicated hyperblock region formation and branch prediction using two branch prediction architectures: a BTB with a 2-bit counter, and a BTB with profile-based direction prediction. Over a subset of SPEC92 benchmarks and UNIX utilities, they showed a reduction in branch mispredict rate of 56% using hyperblock regions. Their work avoids the issue of having to predict predicated branches in regions by ensuring via hyperblock formation that only very infrequently taken branches are left in predicated regions.

Tyson [18] utilizes predicated execution to optimize short forward branches, showing that these constitute a significant percent of both integer and floating point branches and have relatively poor prediction rates. He shows up to a 30% reduction in misprediction rate for the SPEC92 benchmark suite – noting that most of the reduction comes directly from the branches translated into predicate defines that no longer require prediction. Tyson presents results for a region formation method that does not contain any changes in control flow. In addition, he examines an idealized region formation that allows any type of branch to remain in the predicated region, but states that it is unclear how to predict them. We assume for the results in [18], that branches left in these idealized regions are only being predicted when their guarding predicate is true.

2.3 Including Predicate Information in Branch Prediction

In [5], August et. al. presents a modified branch prediction architecture called the *Predicate Enhanced Prediction* (PEP) architecture that incorporates predicate information into a local per-branch prediction scheme. They elaborate on one of the problems of region formation by showing how the transformation of an unbiased branch into a predicate define could cause a previously predictable branch to become unpredictable. Their technique focuses on exploiting the relationship between a given predicate define and a branch guarded by that predicate to recover the original prediction pattern for that branch. This is accomplished by storing the guarding predicate register number of the branch instruction in the BTB. Additionally, two local histories are stored in the BTB entry – one associated with the branch behavior when the guarding predicate is true, and one when it is false. The theory is that "true history" should be used and updated with the same pattern that the original branch accessed – since it will be used if the branch's guarding predicate is true. The "false history" should be used when the branch's guarding predicate is false – hopefully producing a prediction of "not taken". However, realistically, whatever value is currently stored in the predicate register file when a branch is fetched is used to choose between histories. In cases where the predicate define guarding a branch has been issued to the pipeline, but not yet resolved, one may access the "false history" even when the branch's guarding predicate eventually resolves to true. Conversely, one may access the "true history" even when the branch' guarding predicate is false if a predicate value from some previous part of the code set that predicate true and the predicate define that produces a value for the current branch has not yet committed.

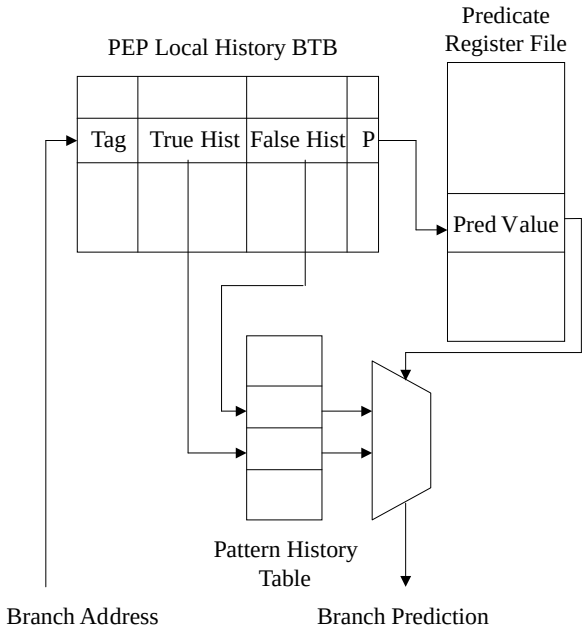


Figure 1: High level design of the Predicate Enhanced Prediction Architecture.

Figure 1 shows a schematic of the PEP branch prediction architecture. A branch prediction takes 2 serial table lookups. The first lookup accesses the BTB providing the guarding predicate associated with the branch. Then another lookup is made in the predicate register file to find the currently available predicate value. The predicate value is then used to choose between the predictions found by indexing a 2-bit pattern history table using the two histories from the BTB. We assume that the local histories are speculatively updated at fetch and correctly recovered on a branch misprediction.

Mahlke et. al [13] examined a new use of predicate registers for collecting information to assist in branch prediction via compiler synthesized information. They proposed new compiler techniques for statically examining register values to produce a dynamically executed function that would

help guide branch predictions. They use predicate registers to hold the result of this dynamically executed function, then this predicate value is used in a modified version of the prepare-to-branch instruction that precedes a branch in their architecture.

Klauser et al. [10] investigated predicated region formation for simple branch hammocks (`if-then` or `if-then-else` constructs only). They examined updating the global history register in the branch prediction hardware with predicate define information. They found only minor to no improvement from updating the global history register because they have no predicated branches in the regions that were formed. In comparison, our approach leaves unbiased branches in the predicated region, and these branches can benefit directly from having the predicate defines update the global history register.

3. PREDICATED REGION FORMATION

Since the cost of predication is directly tied to the cost of falsely guarded instructions executed in a region, intelligent region formation is paramount. At the same time, the benefit that can possibly be derived from predication is directly tied to the original negative impact of the hard-to-predict branches removed by predication. A clear starting point in evaluating the impact of predication is to form regions starting at the most frequently mispredicted branches, with the hope of greatly reducing the number of mispredictions. This would result in (1) not having to predict these very hard-to-predict branches and (2) the removal of frequent and "poorly behaving" entries from the branch prediction hardware, which can reduce destructive aliasing among the remaining branches. However, not all of these important hard-to-predict branches allow for region formation as simple as if-then-else-join conversion. For example, when analyzing branch mispredictions in the SPEC95 benchmark `go` we find several issues that complicate region formation. Most notably, there are several returns that are reached along frequent paths from the most hard-to-predict branches. We find the need to include these return statements in predicated regions if we are to affect any change in the branch misprediction rate of `go` via predication.

3.1 Our Region Formation Algorithm

Our region formation algorithm starts from a list of hard-to-predict branches that we target for translation to predicate define instructions. For the experiments presented here, we start from a list of the top 10% most frequently mispredicting static branches in each benchmark. Original mispredict values are gathered with a baseline Meta Chooser predictor [9] which is detailed in Section 4.

For a given hard-to-predict branch, we walk the control flow graph following the branch incorporating basic blocks into the predicated region. We continue adding successor blocks in a breadth-first fashion until we reach a depth of five basic blocks from the hard-to-predict branch. If, while walking, we encounter another member on the list of hard-to-predict branches, the depth count along that path is reset to zero. This method attempts to target the most frequently mispredicting branches for removal and provides sufficient quantity of post-branch work to overlap the execution of the branch converted to a predicate definition in the pipeline. Additionally, this method provided sufficient scope for our scheduler to investigate a range of region schedules, as will be discussed in Section 7.6.

If a block in the region originally ended in a branch and both of its control flow successor blocks have also been included in the region, then the branch is translated into a predicate define and the successor blocks are assigned the appropriate guarding predicates. If either of the block's successors were not included in the region, then the branch becomes a region branch.

There are additional measures we use in controlling region formation. First any successor block that is reached less than 10% of the time the region is entered is excluded from the region. This keeps cold blocks from unnecessarily bloating the region with infrequently useful work. Second, any block ending in an indirect branch or return automatically stops region formation along that path. Finally, any branch with a successor that has already been included in a previously formed region (such as `(c<a)`), stops region formation along that path.

3.2 Issues with Region Branches

Consider the code in Figure 2(a). Assume that the branch `(a>b)` is a hard-to-predict branch we would like to replace with a predicate define. Applying our hard-to-predict region formation algorithm can produce the region in Figure 2(b). Statement `(b<=0)` is translated into a predicate define because its successors are reached more than 10% of the time we enter the region.

In our example, branch `(c<a)` (very predictable in the original code) is left as a region branch because its taken successor was already incorporated into a region. This leaves only the fall-through successor in the region. Though this is a highly unbiased branch it is very predictable in the original code because of its correlation with branch `(a>b)`. One of the most important aspects of this region is the presence of branches within it with one or more targets outside the region. These branches, like all instructions in the region, are tagged with guarding predicates. These branches will be fetched regardless of the value of their guarding predicate, and at commit, their effect (i.e. whether they are taken or fall-through) is contingent on both their condition (in the case of a conditional branch) and the value of their guarding predicate. In cases where the guarding predicate is false, regardless of branch condition, the branch is not taken.

As can be seen in this example, some always-taken branches (e.g., `return FALSE`) are transformed via the predication process into branches that must be predicted. Though they are still the same type of branch, whether or not they should actually be taken is contingent on whether their guarding predicate is true or false (i.e. whether their path is live or spurious). Hence, these branches now need to be predicted similarly to a conditional branch during fetch. In addition, conditional region branches are now dependent on a combination of conditions when they are predicted. While the branch condition still determines when the branch should be taken, if a conditional branch is predicated on false, it will be treated as a not-taken branch.

In general, all region branches will need to be predicted more frequently than they were in the original code. When we enter a region of predicated code we fetch all instructions down all paths, so we will be predicting branches some number of times more than their actual path of execution is being followed. In Figure 2(b) branch `return FALSE` will be fetched, predicted, and have to be predicted 100K times,

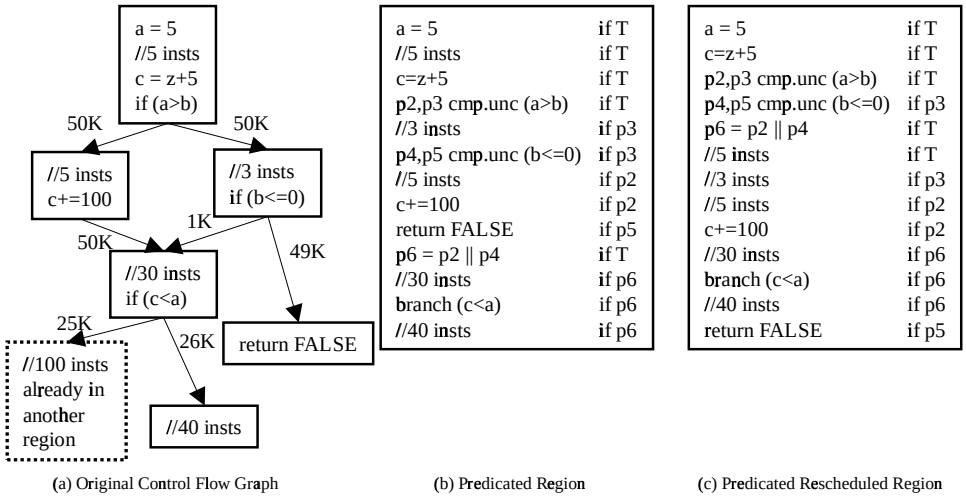


Figure 2: Region formation example that leaves unbiased, possibly predictable branches in the region. Solid boxed basic blocks are formed into a predicated region, the dotted basic block is not included in the region. In this work, we use abbreviated EPIC branch representation to show the condition evaluation as part of the branch. In the original code, branch (a>b) is our targeted hard-to-predict branch. Branch (b<=0) is highly biased and highly predictable. Branch (c<a) is not highly biased, but also originally very predictable given knowledge of the behavior of branch (a>b).

approximately twice as frequently as in the non-predicated code. This impacts the predictability of these instructions and can significantly increase the accesses to the branch prediction hardware.

All of these issues mean that, when using a traditional branch prediction architecture, the region branches in Figure 2(b) suffer from misprediction migration. This is because branch (c<a) and the return branch are both guarded by predicates defined by what was the unpredictable branch (a>b). Therefore, these branches ((a>b) and "return FALSE") are harder to predict using a traditional branch prediction architecture. Branches like the return branch should benefit from a local predicator utilizing predicate information as long as the define of predicate P5 can be scheduled sufficiently before the return. The branch (c>a) which may be predictable based on correlation with (a>b) should benefit from a global history scheme that can incorporate the information produced from the predicate definition of (c>a) even though neither P2 nor P3 is the guarding predicate of (c<a).

4. BASELINE BRANCH PREDICTOR

Our baseline branch predictor is a Meta Chooser [9] style predictor pictured in Figure 3. For the results in this paper, we simulated a 4K entry local, global, and chooser tables using a 12 bit global history register. This type of branch predictor takes advantage of both per-branch local history as well as recent path global branch history in making accurate predictions. This predictor uses the global table to make a prediction in a single cycle, which is squashed and updated if the local prediction made in the next cycle is selected by the chooser.

4.1 Baseline Meta Chooser Predictor

When using the baseline Meta Chooser predictor for predicated code, all region branches still speculatively update the speculative global history register during fetch. One difference in predicated regions is that all direct region branches (e.g., unconditional, returns, etc.) are now treated as branches that need to be predicted as taken or not-taken. Therefore, these branches update the global history register and obtain their direction prediction from the Meta Chooser predictor. For example, a predicated return branch instruction inside of a region determines that the next fetch PC would either be (1) the top of the return stack (taken), or (2) the fall-through PC (not-taken) based upon the direction prediction of the Meta Chooser. Conditional region branches use the Meta Chooser as they do in non-region code to produce a branch prediction. However, the Meta Chooser branch prediction really represents the *combination* of the guarding predicate and the evaluation of the conditional expression when predicting a branch.

In the baseline Meta Chooser predictor, all region branches speculatively update the speculative global history register and the local history register when they are fetched. The update the 2-bit state counters when the branch commits *even if guarded on a false predicate.* Falsely guarded branches are treated as if the branch evaluated to not-taken, and the branch state for a branch guarded on a false predicate will be updated as not-taken.

5. SQUASHING FALSE BRANCHES

The first predicate aware branch prediction architecture we propose uses guarding predicate knowledge to completely squash the prediction of falsely guarded branches. This is a modification of the use of predicate knowledge as defined by August et al. [5]. Their Predicate Enhanced Prediction (PEP) architecture uses the value of a branch's guarding predicate to choose one of two local history registers to use in predicting the branch – channeling branch predictions where the guarding predicate is known to be true to one 2-bit predictor, and those with false or yet-to-be defined guarding predicates to a different 2-bit predictor.

The *Squash False Path* (Squash-FP) architecture we pro-

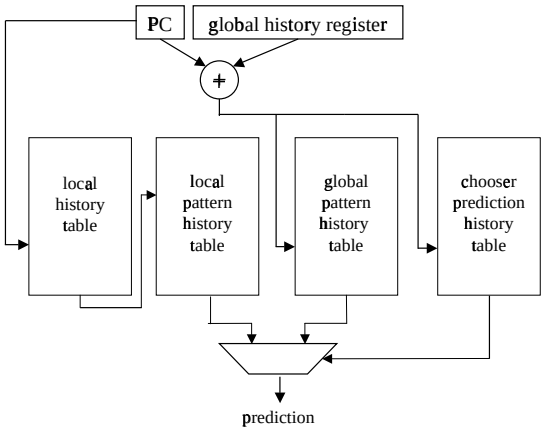


Figure 3: Our baseline Meta Chooser branch prediction architecture. It contains a local history predictor, global history predictor, and a 2-bit chooser table to choose between the two predictions.

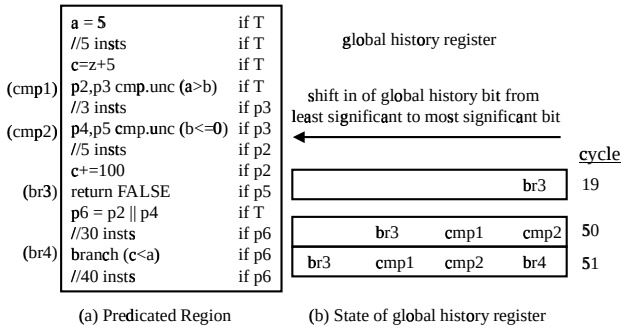


Figure 4: Update of the Meta Chooser global history register when enabled with predicate update. Note that branches update the global history register in fetch, while predicate defining instructions update it in writeback. This causes a "re-ordering" of information in the global history register as compared with instruction fetch ordering. For the purposes of this example, we show a fixed 20 instruction delay from predicate define fetch to resolve. In our results updates occur based upon the resolution latency for individual predicate define instructions.

pose stores the predicate register number that is guarding each branch in the branch's BTB entry. During a prediction the predicate register is looked up in the register table, and the lookup returns not only the value of the predicate, but also if the predicate has any outstanding definitions in the pipeline. A predicate register that has no outstanding definitions in the pipeline is said to be *resolved*. If the predicate is resolved and it evaluates to false, then we accurately predict not-taken for the branch. If the predicate evaluated to true or it was not resolved, then the default predictor is used. Not only will this give 100% prediction accuracy to those falsely guarded branches whose guarding predicates are resolved by the time they are fetched, but it will reduce contention in the tables for the remaining predictions.

To provide this prediction, we rely upon the register lookup to tell us if the latest definition has written to the register, or whether an instruction in the pipeline has yet to produce its value. This information is provided in traditional architectures to determine if a bypassed value from the pipeline should be used instead of the register file value when executing an instruction. We use this same information already provided by processors to tell if the predicate is resolved or not as described above.

6. GLOBAL UPDATE PREDICTOR

We will now examine incorporating predicate information into the global history based branch predictor. Our *Predicate Global Update* (PGU) predictor architecture stores the predicate result from predicate define statements in the speculative global history. When a predicate define instruction finishes its execution, it shifts the result of its condition evaluation into the lowest bit of the global history register in exactly the same way the result of a branch is shifted into the global history register. However, for predicate defines, this occurs when the predicate define value is known in the writeback stage. Since the predicate define instructions perform a *delayed update* of the speculative global history register, the ordering of the global history information will be different than in the original non-predicated program. A delayed (out-of-order) update of the global history register results in very accurate predictions as long as branch/predicate define ordering is similar throughout execution.

For a branch to benefit from predicate information stored in the history register, the predicate update has to occur before that branch is fetched and predicted. This can be addressed by compiler scheduling of the predicate defines as early as possible in the predicated region combined with scheduling the region branches as late as possible. This comes at the cost of executing additional non-true path instructions, since we exit the region later when a branch is taken out of the region.

Region branches (both spurious and true-path) update the PGU predictor in exactly the same fashion as the Baseline Predicated predictor. Both are predicted and speculatively update the global history register in fetch.

6.1 Speculative Global History Update

Figure 4 shows an example of how the predicate update predictor would update the global history register. For this example, we assume a 20 instruction delay from the fetch of a predicate define until it resolves in the writeback stage where we update the speculative global history register. The actual update would occur at varying times for different predicate define instructions depending on their latency from fetch to writeback.

Using the predicate update branch predictor, the speculative global history register is updated with the values of predicate registers `p2` (cmp1) and `p4` (cmp2). The predicate defines formed via our region formation process all define two complementary predicates, and we model the architecture such that it updates the speculative global history register with the value of the first predicate. The IA-64 architecture supports a variety of predicate define instructions that can define up to two predicates using many different boolean combinations [2]. Examining how to use other IA-64 predicate define instructions and their interaction with the predicate update predictor is an area for future work.

As can be seen in Figure 4 our predicate update predictor achieves its goal of incorporating the important history of

statement (a>b) into the speculative global history register. When (c<a) is fetched, the second most recent bit of history in the global history register (cmp1) helps determine the correct prediction for (c<a). This alleviates the problem of misprediction migration that would otherwise manifest for this branch without a predicate update branch predictor.

Using the schedule in Figure 4, the region branch return FALSE is not able to benefit from the predicate information in the global history, since global history from neither of the cmps is updated in time. A possible solution to this issue is to make predicate region code scheduling aware of the correlative behaviors between predicate defines and branches. A conservative solution to the problem is to schedule predicate defines as early as possible in a region while also scheduling branches as late as possible in the region. Figure 2(c) shows a rescheduling of the region in (b) where the number of intervening instructions between predicate define (c<a) and branch return FALSE is increased from 9 to 87. This allows the update of the global history register by the predicate define to complete before we fetch and predict the branches that are correlated with it. Using the region schedule in Figure 2(c) results in a global history register of {cmp1, cmp2, br3, br4}, and allows br3 to benefit from having cmp1 in its global history register during prediction. The optimal schedule would not move predicate defines as far away from branches as possible, but rather, just far enough to allow the predicate defines to update. For example, in the code shown in Figure 2(c) since we modeled a 20 instruction delay for predicate defines, the return FALSE instruction would be just as predictable if scheduled before the 40 instructions from block P5. This would reduce wasted dynamically executed instructions in the cases where return FALSE is taken.

6.2 Recovering the History State

An important topic in branch prediction is recovering the prediction history state after a branch misprediction. Since we are updating the global history register with branches in the fetch stage and predicate define instructions in the writeback stage, it is key that the updates to the speculative global history register occur in a consistent order for a given trace through the program's execution. In addition, we need to be able to correctly recover the speculative global history register in the case of a branch misprediction.

Our architecture uses a Speculative History Queue (SHQ) [15, 16] to hold the state of the global history register, so it can be restored on a misprediction. The SHQ stores the full speculative global history register into a queue each time a branch is predicted. When a branch is mispredicted the global history register for that branch is restored from the SHQ. The last bit in the history register representing the mispredicted branch is inverted, and this becomes the new speculative global history register used for prediction for the next fetch. For example, if branch br4 mispredicts in Figure 4, the speculative global history register will be restored to {br3, cmp1, cmp2, !br4}, correctly keeping track of cmp1 and cmp2.

In using the SHQ architecture as defined in [15, 16], there is a very small window in which a predicate define may not make its way into the SHQ, so that it can be restored after a misprediction. If the predicate definition resolves between the time a mispredicted branch is fetched and the time that branch triggers a misprediction, it will make its way into the speculative global history register, but not into the SHQ. This is because only a branch prediction will insert the speculative global history register into the SHQ. In this situation, the predicate define information will be lost if there is a misprediction before the next branch is fetched. We modeled this in our simulation results in the next section. The SHQ architecture could prevent the loss of this information by inserting into the restored global history register after the branch misprediction any predicate defines that would have been lost. Examining such an enhancement is left for future work.

7. EXPERIMENTAL EVALUATION

7.1 Methodology

We gather results for a subset of the SPEC95 benchmarks (go, gcc, m88ksim, and ijpeg), SPEC92 li, as well as two other benchmarks. Dot is a project from AT&T for plotting graphs, and gs is a run of ghostscript translating a paper from postscript to jpeg format. We chose these benchmarks because they have a reasonable number of mispredictions. We used the same input to the applications to generate the profile to guide region formation and to gather the misprediction results via simulation.

To gather our results we use a predicated form of the Alpha Instruction Set Architecture (ISA). We used the Alpha ISA, adding a predicate guarding register to every instruction, and we added predicate compare instructions to the ISA to define the predicates as shown in the examples earlier in this paper. We used this modified ISA to build predicated regions and to simulate the predicate scheduled code.

To conduct our experiments we used both ATOM [17] and SimpleScalar 3.0a [6]. We used SimpleScalar to calculate the average fetch to writeback latency for each branch instruction. This latency represents the number of cycles it takes to complete execution of a branch instruction from the time the branch instruction is fetched. We calculate this latency in SimpleScalar for each branch instruction and use it to build our own scaled down pipeline level simulator in ATOM in order to simulate the full execution of a program. For the SimpleScalar runs we simulate an 8-wide issue machine with a 128-entry RUU. The L1 data cache is 64K 4-way associative, L1 instruction cache is 32K 2-way associative, and we use a unified 1 MB 4-way L2 cache. The L1 miss and L2 hit latency is 12 cycles, with 120 cycle latency for an L1 and L2 miss. The minimum branch misprediction penalty is eight cycles, and we use a 32-entry return address stack for predicting return instructions.

To generate the predicate regions we used ATOM to profile the entire execution of the program to find the hard-to-predict branches. We then used the program analysis features of ATOM to provide an intermediate representation of the binary to schedule. We use this IR to form our predicated regions, as described in section 3, resulting in a new predicated representation of the binary. We also used ATOM to build a pipeline level branch simulator using the branch latencies from SimpleScalar, and simulate the predicated representation of the binary. This allows us to simulate the complete execution of a program, modeling the important branch latencies. We use the branch and predicate define latencies to model the out-of-order nature of predicate define updates to the speculative global history register, which occur once the predicate instruction resolves. We also use this to model the effect of recovering the branch

prediction state on a branch misprediction at the cycle the branch instruction finishes execution.

We evaluate our architecture by examining the improvements in branch misprediction rates, which are all normalized to the number of branch mispredictions in the original non-predicated code. That is, the misprediction "rate" is calculated as the number of mispredicts divided by the number of branch predictor accesses in the *original* execution of the program code. This allows us to look at one consistent metric as the number of branch predictor accesses will change with the predicated code. In addition, the miss rates we show include the misprediction rate for all branch types. This includes conditional branches, returns, unconditional, procedure calls, and indirect branches. In addition, we show the percent increase in instructions executed for the hard-to-predict predicate regions formed.

7.2 Baseline Prediction Results

We start by examining the percentage of mispredicted branches without using any predicate information to update the branch predictor. The first three bars in Figure 5 show the original mispredict rates using only local prediction, only global prediction, and the Meta Chooser (combination of local and global). We simulated a 4K entry local history, local pattern, global pattern, and chooser tables using a 12 bit local history and global history registers.

We show that the original Meta Chooser predictor performs better than either a predictor using just local (per-branch) information or one using just global information. The fourth bar in the graph shows the percent of mispredicted branches that occur after applying our region formation algorithm and predicating the hard-to-predict branches.

The results show that substantial reductions in miss rates are achieved for `go` (22% down to 15%), `ijpeg` (8.5% down to 2.5%) and `m88ksim` (3% down to 1%) using the hard-to-predict region formation method with a traditional branch predictor.

For the other programs, the results appear to show that we are not successful in attacking the misprediction problem with our hard-to-predict region formation approach. In reality (as we'll see in the detailed breakdown of mispredictions to follow) we are removing a large percent of mispredictions by translation of hard-to-predict branches to predicate defines, but the remaining region branches become much harder to predict using the baseline predicated Meta Chooser. Misprediction migration leads to disappointingly high misprediction rates, so much so that in `dot` and `li` very little decrease in overall misprediction rate is seen.

7.3 Predicate Global Update Predictor Results

The final four bars in the Figure 5 examine various methods of using predicate information in a Meta Chooser predictor and compare them to an idealized execution of the predicated code, where only branches executed on the true paths in regions have to be predicted. The details of each implementation follow. All predictors use a 4K-entry local history table to store local histories, a 4K-entry 2-bit pattern history table for local predictions, a 4K-entry 2-bit pattern history table and a 12-bit global history register for global predictions.

- Meta Chooser PEP: One of two local histories stored for the branch is used, based on the value of the guarding predicate register. The chosen history is the only one updated with the result of the branch. This provides results for the PEP predictor presented in [5].

- Meta Chooser Resolved PEP: We modified PEP to choose between its two local histories based on (1) knowing if the most recent predicate register definition for the guarding predicate is resolved or not, and (2) the value of the guarding predicate. If the most recent definition of the guarding predicate register has not resolved, then the false predicate local history is used to predict the branch. If it has resolved, than either the false or true local history is chosen based upon the value of the guarding predicate. The chosen history is the only one updated with the result of the branch. The results show that concentrating the true path local history on only those branches that have resolved provides a decent reduction in miss rate for a `li`.

- Meta Chooser PGU: The predicate define update of the speculative global history register is delayed until the predicate instruction has resolved. At this time its value is entered into the speculative global history register. Branches that are fetched after this that correlate with this predicate definition will benefit from having it in the global history register.

- True Path Only: For these results we model an environment where only those branches whose guarding predicate is true (i.e. whose path is live) are predicted and update history information. This isolates the effects of predicting branches guarded on a false predicate and provides an indication of what mispredict rate would be expected given the removal of the hard-to-predict branches via predication.

A Meta Chooser with Resolved PEP local predicate update averages a mispredict rate of 4.5% and one with PGU global predicate update averages a mispredict rate of 5%. Only having to predict branches that are guarded by a TRUE predicate, even with no predicate update, results in a miss rate of 2.75% on average.

7.4 The Impact of Falsely Guarded Branches

In Figure 6 we show a more detailed breakdown of the branch mispredicts in the original code, the code after predication, using Resolved PEP-style predicate information to affect local predictions, using the PGU predicate information to affect global predictions, and an idealized world where only true path branches have to be predicted.

The top portion of the original bars show the percent of mispredicts that will be removed from transforming those branches into predicate define statements when applying our hard-to-predict region formation algorithm. These results show that 44% to 91% of the original mispredicts in the programs are removed by if-converting these hard-to-predict branches.

The middle section of the original bars (True Path in Region) shows the mispredicts that come from region branches whose guarding predicate evaluates to true. The False Path in Region shows the percent of mispredicts caused by region branches that are guarded on false. The remaining mispredicts (in black) will lie outside of our predicated regions and will not be *directly* impacted by our predictor modifications,

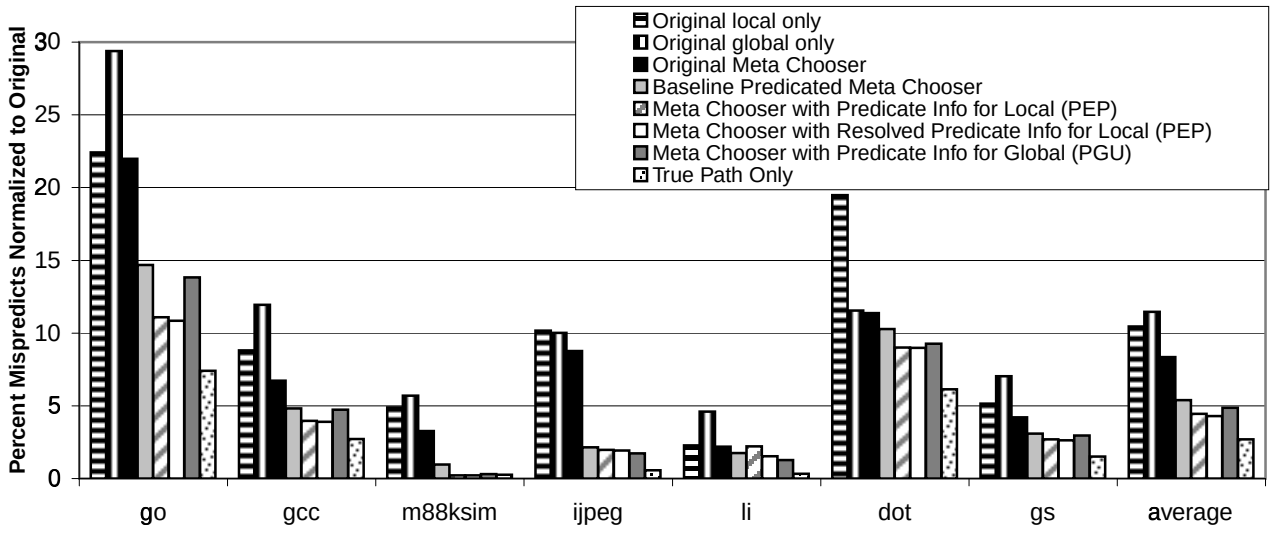


Figure 5: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code.

but may be affected by the global history update from predicated regions.

The number of dynamically fetched branches for predicated code where only the true path branches are fetched is reduced between 23% (for `li`) and 50% (for `ijpeg`). However, when considering the spurious (false path) branches that must be fetched in predicated regions, the number of dynamically fetched branches is not always reduced compared to the original code. Three benchmarks still have overall reductions in dynamically fetched branches (`m88ksim` , `ijpeg`, and `dot`), two have almost the same number of predictions as in the original code (`go` and `gcc`), and the rest see an *increase* in the number of dynamically fetched branches.

In Figure 6, across all the predicated results, we see little or no decrease in mispredicts for non-region branches and a marked increase in the number of mispredicts caused by region branches. Additionally, while false path mispredictions are significant in and of themselves, by comparing true path only results with the other predicated results, we see that false path (spurious) predictions and mispredictions have a marked impact on the mispredicts caused by true path region branches.

7.5 Squash-FP to Remove False Branches

In Figure 7 we show various branch prediction schemes used to minimize the negative impact of falsely guarded branches on region branch prediction. Squash-FP uses information from the predicate register file to determine if a branch's guarding predicate is false and then effectively squash the branch by predicting it as "not-taken". The Squash-FP prediction filter is only used if the most recent definition for the guarding predicate has resolved. Squash-FP requires knowing the guarding predicate register for each branch, which we assume is saved per branch in the BTB.

Figure 7 shows six results – 4 using Squash-FP to first filter out the branches that are guarded on false predicates, where the predicates have resolved by the time the branch is predicted. The implementations shown are:

- Baseline Meta Chooser: Uses no predicate information, must predict all region branches.
- Squash-FP Baseline Meta Chooser: Uses guarding predicate information as stored per branch in the BTB. Predict not-taken for branches whose guarding predicate is resolved and has a value of false.
- Squash-FP PEP Meta Chooser: If Squash-FP does not apply to the branch, then use the predicate value from the predicate register file to select between the two per branch local histories. Use this local history if local history is chosen by the meta chooser, otherwise use the default global predictor.
- Squash-FP PGU Meta Chooser: If Squash-FP does not apply to the branch, then use PGU to predict the branch if global prediction is chosen by the meta chooser. All branches update the global history register, otherwise use the default local predictor.
- PEP+PGU Meta Chooser: Uses PEP-style predicate information for local predictions and PGU-style for global predictions.
- Squash-FP PEP+PGU Meta Chooser: If Squash-FP does not apply to the branch, uses PEP-style predicate information for local predictions and PGU-style for global predictions.

Each bar in the graph is broken into five sections indicating the source of the misprediction. The bottom section shows misses from Non Region areas. The next two show misses from true path branches in regions - the stripes are from prediction where the value of the guarding predicate was known to be true at the time of prediction. The top two show misses from false path branches - the stripes are from predictions where the value of the predicate was resolved false (these predictions are squashed via Squash-FP).

The four implementations of Squash-FP show a clear benefit over the other two in that all misses from resolved false

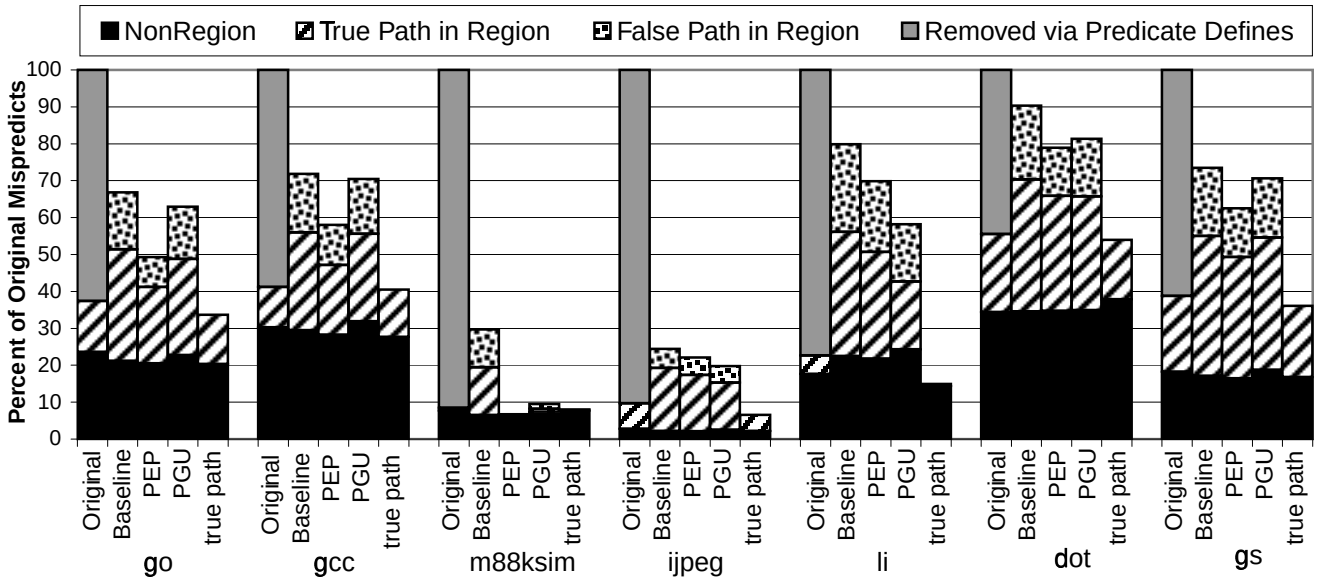


Figure 6: Comparison of the breakdown of locations of mispredicts in the program, normalized to the original number of mispredicts per benchmark. In each set of bars we show information (from left to right) for the original non-predicated code, the baseline Meta Chooser predictor on the predicated code, Meta Chooser using PEP update for local predictions, Meta Chooser using PGU for global predictions, and an idealized execution of only the true path of the predicated code.

path predictions (dark stripes) are removed. Some benchmarks like `gcc` and `gs` do better when incorporating predicate information into the local predictor. Some benchmarks do better when incorporating predicate information into the global history register. The latter can be particularly beneficial when the full guarding predicate of a branch is not resolved at prediction. `li` shows significant benefit from PGU update which we attribute to its very low percent of resolved predicates (as discussed in Section 7.6). Some benchmarks like `go` , `gcc`, and `m88ksim` achieve as much benefit from using predicate information to just reduce resolved false path predictions as from further predicate usage in either PEP or PGU. On average, PEP sees very little improvement with Squash-FP, PGU sees .6% improvement with Squash-FP, and PEP+PGU sees .1% improvement. A Baseline predictor that only uses predicate define information to do Squash-FP improves its misprediction rate by a full 1% with this spurious branch detection optimization.

7.6 Increase in Executed Instructions

We applied our region formation algorithm in section 3 to form predicated regions for the top 10% most frequently mispredicting branches in each benchmark. For the SPEC95 program `go`, 87% of all mispredicts in the program can be attributed to the top 10% most frequently mispredicting static branches. Additionally, we aggressively scheduled predicated regions trying to separate predicate defines and branches at the cost of execution of extra code. As stated earlier, we did not try to limit the increase in executed code since our study focused on examining the maximum reduction in branch misprediction rate achievable using the techniques presented in this paper. Dynamic instruction counts of the predicated versions of the codes increased across a wide range from 10% to 94% as shown in Figure 8.

The regions we formed provided a large range of exploration with regards to predicate define and branch scheduling placement. Scheduling of branches and the predicate defines that guard them is paramount as most of the predicate update predictor techniques evaluated rely on information from "resolved" branches – ie, those branches whose guarding predicates have been defined by the time the branch is fetched. Our region formation lead to widely ranging numbers of branches whose guarding predicate is resolved: in `li` only 7.5% of all region branches have their guarding predicate resolved at fetch, while in `m88ksim` all region branches have their guarding predicate resolved.

In future work, we will be examining the effects that this region formation has on performance and refining our region formation algorithm to tradeoff the increase in code with the reduction in branch miss rate.

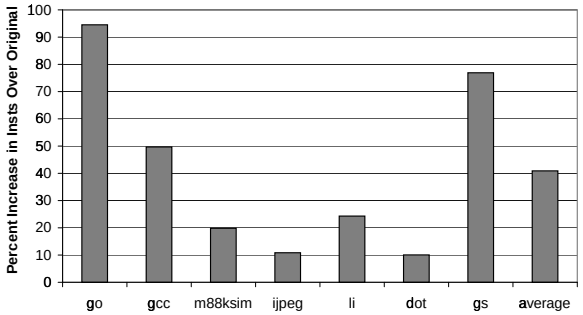


Figure 8: Increase in dynamic instruction count from our hard-to-predict predicate region formation and scheduling techniques.

7.7 Variability of Predicate Define Updates

Due to the delayed update of predicate define information, any scheme which utilizes predicate define information

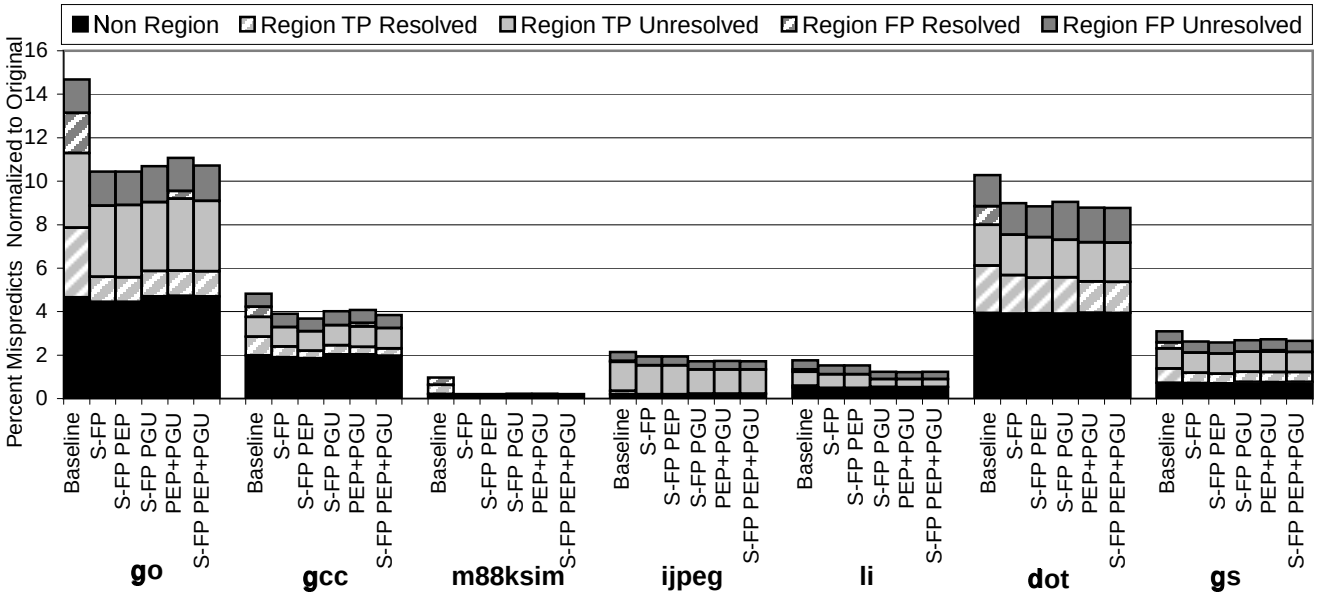


Figure 7: Change in misprediction rate, normalized to the number of predictor accesses in the original unpredicated code. Columns 1 and 5 of each group are not optimized with Squash-FP, the rest show the benefit gained by removing resolved false path predictions.

in conjunction with a global history register is problematic given variability in the latency of predicate define instructions.

With the PGU scheme, if a predicate define uses an operand whose load occasionally misses in the cache, then some dynamic occurrences of branches fetched shortly after the predicate define will see "different" global history registers. In cases where the load does not miss, the predicate define may reach commit and update the global history. In cases where the load misses, that predicate define may not update the global history, yet other fetched branches may update the global history. This variability in ordering of global history update will have a negative effect. Our results currently use a fixed length predicate define latency determined for each branch from SimpleScalar. The only differences in global history update sequences we see occur when a branch mispredicts and additional cycles are available (from the mispredicted path) for predicate defines to update the global history register. For Squash-FP, we always update the global history of a region branch, even if its predicate is resolved false and its prediction is squashed. This way, while our ability to determine that a branch is guarded on false might differ based on system effects, the global history register entries will not suffer from variance.

While this effect might be relatively small in an in-order processor like the Itanium, the issue will certainly be more noticeable in an out-of-order implementation. One possibility would be to fix a delay time for each static predicate define. Then, as the predicate define was fetched, an entry would be made in a structure that would be decremented every cycle – effectively acting as a timer. When a predicate define's timer went off, if the define had resolved, the resolved value would update the GHR. If not, the GHR would update with false. This would guarantee fixed update of the GHR and would allow the compiler flexibility in the update of the GHR with predicate define information. We are examining the performance of this design as part of future work.

7.8 Comparing PEP and PGU

The PEP predictor uses the predicate information to choose which local history prediction to use, whereas our PGU predictor includes the predicate information into the global history register.

For the results presented, we examined both architectures using roughly the same area. Comparing the complexity of these two designs in terms of access time, the PEP architecture will be difficult to implement in a single cycle. The critical path of the PEP architecture requires a two table lookup to perform a conditional prediction, whereas the PGU architecture requires only a one table lookup to perform its conditional branch prediction. The PEP architecture first needs to look into the BTB to find out which predicate corresponds to the branch and the two local histories. It then looks up, in parallel, the predicate register file to obtain the latest value for that predicate and the 2-bit predictors for the two local histories. Only then can it choose its prediction using the result of the register file lookup. In comparison our PGU architecture performs only one table lookup by using the global history register to index into a 2-bit table of counters, similar to existing branch prediction architectures. The reduced complexity and access time of the PGU design makes it an attractive option for implementing a predicate-aware branch predictors.

8. SUMMARY

Predication allows hard-to-predict branches to be removed and replaced with predicate defines, which do not have to be predicted. In order to effectively reduce branch predictions, predicated region formation must focus on the most offending hard-to-predict branches. To do this we found that we had to allow unbiased, though originally predictable,

branches to reside in predicated regions. Therefore, we developed a hard-to-predict region formation algorithm targeted at removing many of these hard-to-predict branches, while allowing some unbiased, predictable branches to remain in the predicated region. On average, predicate region formation reduced the branch mispredict rate from from 8% to 5.5% across the benchmarks when using our hard-to-predict region formation.

Without any modification to branch prediction hardware, region branches become a major problem in achieving the reduced branch misprediction rates we expect from predicated codes. The ability to accurately predict these region branches is hindered by their increased dynamic occurrence, their new prediction pattern based on their guarding predicate dependences, and the fact that information from predicate define statements is no longer available in the global history register.

To address this problem, previous work has investigated augmenting local per-branch prediction schemes with guarding predicate information. We propose a complementary Predicate Global Update Branch Predictor architecture to improve the prediction of region branches with the goal of reducing misprediction migration. Our Predicate Global Update Branch Predictor allows predicate define statements to provide correlative information to the branch predictor state by updating the speculative global history register at writeback. We model updating the speculative global history register out of order using predicate define statements, and recovering the state of the branch predictor on a branch misprediction.

As much as possible, we schedule regions to allow predicate define information to be utilized by both of the predicate sensitive prediction schemes.

Finally, we propose a direct approach to the problem of branches whose guarding predicate is false using Squash-FP. Squash-FP achieves 100% prediction accuracy for spurious branches whose guarding predicate definitions have resolved by the time the branch is fetched. These falsely guarded branches should always predict "not-taken". This technique uses a branch-to-guarding predicate identification scheme via the BTB similar to PEP. However, it takes advantage of bypass information stored in the architecture pipeline to determine when branches are guaranteed to be spurious. This technique also has the benefit of reducing contention in the branch prediction tables by no longer using predictor state that was previously used by false path branches.

We show that, even alone, the Squash-FP method of utilizing predicate define information achieves a sizable reduction in branch mispredictions (ranging from .5% to 4.3%). This method is arguably the simplest and fastest predicate update modification to current branch prediction architectures. Squash-FP can also be employed with PEP, PGU, or a Meta Chooser predictor utilizing both PEP and PGU. The benefit measured with these techniques is modest on the average, but individual benchmarks experience significant improvements.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded by NSF grant No. CCR-0073551, and a grant and equipment donation from Intel Corporation.

9. REFERENCES

[1] Intel Corporation: Itanium Processor Architecture. http://www.intel.com/design/ia-64/index.htm.

[2] *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*. January 2000.

[3] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, July 1998.

[4] D. I. August, W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[5] David I. August, Daniel A. Connors, John C. Gyllenhaal, and Wen mei W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *The 3rd International Symposium on High-Performance Computer Architecture*, pages 84–93, 1997.

[6] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[7] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th Annual Intl. Symp. on Microarchitecture*, December 2001.

[8] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[9] R.E. Kessler, E.J. McLellan, and D.A. Webb. The alpha 21264 microprosessor architecture. In *International Conference on Computer Design*, December 1998.

[10] Artur Klauser, Todd Austin, Dirk Grunwald, and Brad Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pages 278–285, 1998.

[11] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 217–227, December 1994.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[13] Scott A. Mahlke and Balas K. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 153–164, 1996.

[14] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[15] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. In *IEEE Transactions on Computers*, April 2001.

[16] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction Level Parallelism*, January 2000.

[17] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. Association for Computing Machinery, 1994.

[18] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, November 30–December 2, 1994.

Fracture Mechanics on the Intel® Itanium™ Architecture

(A Case Study)

Gerd Heber and Andrew J. Dolgert
Cornell Theory Center
Rhodes Hall
Ithaca, NY 14853
heber@tc.cornell.edu
ajd27@cornell.edu

Maxim Alt
Intel Corporation
SC12–411
3600 Juliette Lane
Santa Clara, CA 95054–1513
maxim.alt@intel.com

Karen A. Mazurkiewicz and Lynd Stringer
Intel Corporation
CH7–401
5000 W. Chandler Blvd.
Chandler, AZ 85226–3699
karen.a.mazurkiewicz@intel.com
lynd.stringer@intel.com

ABSTRACT

We optimized our fracture mechanics code to achieve a 3.4-fold speedup on Itanium processors. The computational core of the code is a linear equation solver using a preconditioned conjugate-gradient method. The dense and sparse matrix-vector computations are both floating point and memory bandwidth intensive. We found that the EPIC architecture depends heavily on the compiler to find instruction level parallelism and thus to achieve maximum performance.

1. INTRODUCTION

The Intel® Itanium™ processor is the first commercially available implementation of Intel's EPIC technology. A natural question is: given a legacy code, what optimization is necessary to achieve reasonable performance on an Itanium processor-based system.

In this paper, we describe our effort to tune a fracture mechanics code, Crack Propagation on Teraflop Computers (CPTC) [3, 4]. The CPTC software is a joint effort of engineers, computer scientists, and numerical analysts to simulate the growth of arbitrary cracks in 3D solids based on linear elasticity. Figure 1 is a high-level view of the interaction of major components in CPTC. After creation of an initial model, the spatial domain is decomposed through volume meshing into simple elements such as tetrahedra or hexahedra. Thereafter, a discretized version of the underlying (continuous) elasticity equations is derived using the finite element formulator. After solving the equations, a fracture analysis is performed to predict the growth of cracks and make a prediction for the duration of crack growth. Typically, tens or hundreds of crack growth steps, each involving volume meshing and equation formulation and solving, must be performed for a single lifetime prediction.

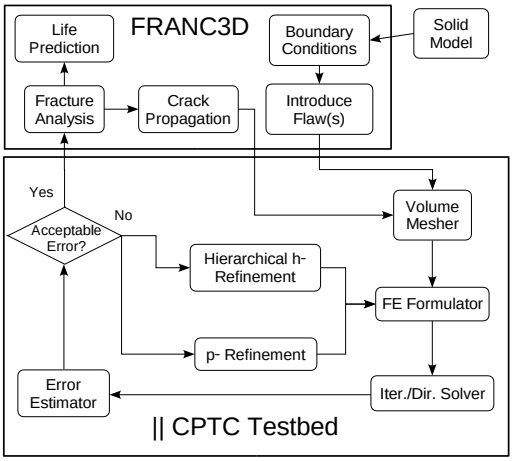


Figure 1: High-level view of the CPTC simulation environment.

CPTC is written in C/C++, and the performance-critical part is the linear equation solver with both dense and sparse matrix-vector computations. While the CPTC software runs on parallel computers with distributed memory using the Message Passing Interface (MPI) [7] API, the focus of our tuning effort was sequential or per-process(or) performance. No changes were made to the MPI code.

The CPTC code is characterized by loops, few branches, and is floating-point and memory bandwidth intensive. The key to maximum optimization was helping the compiler find the maximum instruction level parallelism in the computationally intensive loops. To do this, we changed source code and also compiler flags. To measure progress, we used the assembly language output, high-level optimizer (HLO) report, and software pipelining (SWP) reports. Our optimizations resulted in an 3.4-fold speedup on Itanium processors.

The paper is organized as follows: In Section 2, the computational kernels underlying the solver and a general discussion of the main implementation issues are presented. Section 3 discusses relevant features of the Itanium architecture. Sec-

tion 4 contains some remarks about the tuning methodology. This is followed by Section 5 where we present the key steps of our tuning effort in a condensed form. In Section 6, we summarize the main results and lessons that we learned along the way. Conclusions are drawn in Section 7

2. COMPUTATIONAL KERNELS—A PRIORI CONSIDERATIONS

As mentioned in the introduction, the solver is the most time consuming part in a linear fracture analysis. It uses the conjugate gradient method [11] with global extraction element-by-element (EBE) preconditioning [17]. In the next two sections, we describe two main components, the iterative solver and the preconditioner, and how their computational kernels challenge the architecture.

2.1 The Solver

In CPTC, fracture mechanics is modeled in terms of linear elasticity equations. These partial-differential equations are discretized using the Finite Element Method (FEM). The result is a system of linear equations $Ax = b$. The main characteristics of A can be summarized as follows:

- A is an $N \times N$ matrix where N is large, typically on the order of 10^6.
- A is *sparse*, i.e. it has only $\mathbf{O}(N)$ nonzero entries.
- A is symmetric and positive definite.
- A has a distinct *structural pattern*, e.g. most blocks of three consecutive rows have nonzero entries in the same column positions.

The preconditioned conjugate gradient (CG) method [11] is a standard iterative solver for symmetric and positive definite problems. An extensive suite of iterative solvers, including CG, is part of the Portable, Extensible Toolkit for Scientific computation (PETSc) [12, 14, 13] used in CPTC. The main computational labor per iteration consists of the following:

- One matrix-vector product $y_i := \sum_j a_{ij} x_j$
- The application of the preconditioner (see Section 2.2)
- Two dot products $x \cdot y := \sum_i x_i y_i$
- Three DAXPY type operations $y \leftarrow y + \alpha x$.

The matrix-vector product, dot products, and DAXPYs are very memory bandwidth demanding. There is no temporal locality and the ratio between floating point operations and load/store operations is rather low. This is problematic because of the great imbalance between the available memory bandwidth and the (high) rate of execution in the processor's floating point units. For the matrix-vector product, depending on nonzero structure, spatial locality may be poor and memory latency critical. Toledo's exemplary discussion in [18] demonstrated a combination of blocking, prefetching, and reordering as an effective remedy to make the best use of memory bandwidth. For CPTC, however, blocking and reordering are not directly applicable without major code changes.

2.2 The Preconditioner

The global extraction element-by-element (EBE) preconditioner was first described in [17] and all details can be found there. Here, we restrict ourselves to a brief description of the main idea.

Within the FEM, the global matrix A is the result of an assembly process $A = \sum_e A_e$ where each finite element e (such as tetrahedra or hexahedra in 3D) contributes a small (dense) elemental matrix A_e. Because of the adjacency of these elements, e.g. two elements sharing a vertex or an edge, there is a certain overlap between those contributions. There is a unique submatrix A_e^{ex} in A associated with each finite element. (A_e^{ex} consists of A_e plus the overlapping contributions from adjacent elements. For CPTC, a typical size of A_e^{ex} is 30×30 when using 10-noded tetrahedral elements with 3 degrees of freedom per node.)

The key idea of EBE is to use the family $\left\{(A_e^{ex})^{-1}\right\}_e$ as a preconditioner. The practical implementation has three separate kernels:

1. Since the A_e^{ex} do not change during the CG iteration, the inversion (factorization) has to be done only once as part of the preconditoner setup (using LAPACK's [15] `DPPTRF` routine).
2. For each element in each CG iteration, a *solve* $z = (A_e^{ex})^{-1} x$ must be performed (LAPACK's `DPPTRS`, which amounts to two calls to BLAS' `DTPSV`).
3. For each element in each CG iteration, a *gather* operation must be performed before the solve and a *scatter* operation after the solve.

```
/* gather */
for (i = 0; i < ndof; ++i)
  x[i] = y[pos[i]];
...solve...
/* scatter */
for (i = 0; i < ndof; ++i)
  y[pos[i]] += z[i];
```

There is a vast amount of parallelism in EBE, because the gatter/scatter and solves can be performed independently for all elements. (Typically, there are on the order of 10^6 elements in a finite element mesh). On the other hand, locality may be poor for the gather/scatter operations, the solve (equal to two triangular solves) involves divisions or reciprocals, and the average loop-trip-count in the innermost loop of the triangular solver is small.

3. FEATURES OF THE ITANIUM ARCHITECTURE

Among the novel features of the Itanium architecture (compared to Pentium architecture) that are the most relevant to a CPTC-type code are the following:

- Support for software pipelining, through
  - Predicate registers
  - Rotating registers

Table 1: STREAM (MB/s), `efl -O2/-O3`.

Test	1 Thread		2 Threads		4 Threads	
Copy	782.0	1160.1	1148.9	1160.6	1227.4	1187.1
Scale	772.7	1169.0	1128.7	1169.6	1223.3	1161.2
Add	528.5	1319.0	945.3	1316.5	1388.0	1314.2
Triad	518.7	1339.3	925.4	1337.1	1394.8	1335.4

- Counted loops with zero cycle branch latency
- An issue rate of two `fma` instructions per cycle
- Issue of six instructions per cycle in two bundles of three instructions each.

It remained to be seen to what extent our code would benefit from these new features. The following subsections contain a quick rundown of two traditional architectural measures.

3.1 System Configuration

Hardware

Processor(s): 4× Itanium (C0) @ 733 MHz

Cache: See table 2

Memory: 4 GB RAM, 2×133 MHz FSB, 2.13 GB/s peak bandwidth

Software

OS: Microsoft Windows XP Advanced Server, 64-Bit Edition, Build 2462

SDK: Microsoft Windows Platform SDK Beta 2

Compiler(s): Intel C/C++ and Fortran compilers 5.0.1B-30.2

Runtime: MPI Software Technology MPIPro 6.3, 64-Bit Edition

Profiler: Intel VTune Performance Analyzer 4.5 for Itanium

3.2 Memory Bandwidth

Memory bandwidth appears to be crucial in many parts of CPTC. The STREAM [10] benchmark gives us a rough estimate of how much bandwidth we can expect on our target system. Table 1 shows the results for the OpenMP [9] versions of STREAM with one, two, and four threads, respectively. All results are in megabytes per second with parameters `N=5000000, NTIMES=30, OFFSET=8`. (Technical specifications for the test system can be found in Section 3.1.) For each number of threads, there are two columns. The left column is for the version compiled with `-O2` and the right for the `-O3` version. The sole difference between the `-O2` and `-O3` versions is that in the latter case all STREAM loops are prefetched, which accounts for the higher bandwidth requirements. The figure shows that the memory bus is saturated only when using four processors in the `-O2` version.

3.3 Floating Point Performance

The Itanium processor can issue 2 `fma` instructions (in separate `mfi` bundles) per cycle. On an 800 MHz processor, the theoretical peak performance is 4 floating point ops (= 2 `fma`) times 800 million per second, in other words, 3.2 Gflops. A more realistic estimate for the sparse matrix-vector operations should be based on memory bandwidth. The peak read-only memory bandwidth for the Itanium processor-based system is roughly 2 GB/s. An L2 cache line is 64 bytes, or eight double-precision floating-point numbers. Assuming that there is no locality and that we have to go to memory for each floating point operation, we arrive at 250 Mflops. Over-estimating the memory subsystem and under-estimating the locality in our code, 250 Mflops becomes a rough estimate of the possible performance on our Itanium processor-based system.

4. METHODOLOGY

The process of optimizing changes because of new characteristics of EPIC technology. EPIC puts the entire burden of Instruction Level Parallelism (ILP) detection on the compiler's shoulders. Since the Itanium processor is an in-order machine, all the available parallelism must be present in the assembly. Thus, the compiler's translation from our source code to assembly is critical.

We chose to measure the performance of the CPTC by the per iteration time in the iterative solver. We established a *baseline* by compiling the entire code with the `-O2` flag and running it on a sample problem of about 30,000 degrees of freedom on a single processor. (This is a typical workload for a single processor in a parallel job.) On our test configuration, which is specified in Section 3.1, the execution time per solver iteration was 0.486 seconds. Using the VTune Performance Analyzer, we identified the following hotspots based on percent of execution time:

- 47% in `DTPSV`
- 30% in the (sparse) matrix vector product
- 8% in the gather/scatter operations.

Qualitatively this confirmed our expectations, though it was not clear at this point why, for example, we would spend 50% more time in `DTPSV` than in the matrix-vector product.

The following (simplified) pseudo code shows our methodology after establishing a baseline. The analysis proceeds in loops, but there are a lot of branches in some of them.

```
_hotspots:
  identify new hotspot using VTune;
_change_source:
  make changes to the source code;
_check assembly:
  recompile;
  satisfied = check(assembly output, HLO
                    and SWP reports);
  if (satisfied) {
    /* the assembly output looks okay */
    progress = run test problem;
    if (progress) /* faster */
```

Table 2: Itanium caches.

	Size	Associativity	Security	Cacheline	Method	Latency (Cycles)
L1 (I)	16 KB	4-way	Parity	32 Bytes	Write Through	
L1 (D)	16 KB	4-way	Parity	32 Bytes	Read Only	2
L2	96 KB	6-way	ECC (64-Bit)	64 Bytes	Write Back	6 (INT), 9 (FP)
L3	4 MB	4-way	ECC (64-Bit)	64 Bytes	Write Back	21 (INT), 24 (FP)

```
    create new VTune sample;
  else { /* (unexpectedly) slower */
    retry = analyze changes made;
    if (retry)
      goto _check_assembly;
    else {
      discard source changes;
      goto _hotspots;
    }
  }
} else /* the compiler doesn't get it */
  if (cflags_changed) { /* played with flags */
    reset compile flags;
    cflags_changed = false;
    goto _change_source;
  } else { /* lets change compilation flags */
    change compile flags;
    cflags_changed = true;
    goto _check_assembly;
  }
```

This loop terminates when either the architectural limits have been found, or the performance goals have been met.

Notice we use analysis of assembly code and compiler reports as a common first measure of the code's clarity (to the compiler) rather than as a late optimization step. In fact, we did no hand-coding of assembly during our tuning effort. Assembly code is time-consuming to write and maintain and is unnecessary in most cases beacuse source code changes and compiler flags offer easy, adequate control to achieve high performance.

5. EXAMPLES

In this section, we showcase the optimizations that yielded the largest performance gains in CPTC. We did not conduct the optimizations in this order. Rather, the five of us were simultaneously exploring different parts of the code. The "true story" can be found in Section 6.

5.1 The `DTPSV` Function

The CPTC code uses the CLAPACK distribution [2], which, as described in the release notes, was created using the f2c [5] Fortran-to-C translator. `DTPSV` is a BLAS function that solves systems of equations $Ax = b$ or $A^t x = b$ where A is an upper or lower triangular matrix. There are four core loops in `DTPSV`. (Note that there are also four other loops dealing with increments in the vector x that are different from one. For CPTC, however, the increment is always one.) The loops are discussed in the following subsections. We use the original line numbers in `dtpsv.c` from the Netlib [2] CLAPACK distribution.

In the following discussion we assume that the reader has a reasonable understanding of software pipelining (SWP).

```
130 for (i__ = j - 1; i__ >= 1; --i__) {
131   x[i__] -= temp * ap[k];
132   --k;
133 }

.b1_47:
 {   .mmi
  (p16) ldfd    f37=[r33]          //0:131
  (p16) ldfd    f32=[r2],-8        //0:131
  (p16) add     r32=-8,r33 ;;      //0:130
 } { .mfi
  (p16) lfetch.excl.nt1 [r44]      //1:130
  (p20) fnma.d  f42=f6,f36,f41     //9:131
  (p16) add     r42=-16,r44        //1:130
 } { .mib
  (p24) stfd    [r41]=f46          //17:131
        nop.i   0
        br.ctop.sptk    .b1_47 ;;  //1:130
}
```

Figure 2: Loop 1. Disambiguation permits the compiler to software pipeline.

Reference [16] is a good introduction and Volume 1 of reference [1] has a nice account of the Itanium processor's support for SWP.

A note on how to interpret the assembly code

Below, we show a few code examples with their assembly output. The compiler dumps key information about the instruction schedule in the form of comments (the text after the `//`) into the assembly file. The comments are of the form

```
(predicate) instruction (;;) // cycle : source line
```

Note that the cycle number is zero-based. For example, assuming that the following `fnma` instruction is part of a software pipelined loop, it would be issued on cycle 9 in stage 5 of the software pipeline (stage one is controlled by `p16` [1]) and corresponds to source line 131.

```
(p20) fnma.d  f42=f6,f36,f41     //9:131
```

5.1.1 Loops 1 and 2

Because of the loop-carried memory dependencies in lines 131 and 173, the loops shown in the upper parts of Figures 2 and 3 are not software pipelined when compiled with `ecl -O2`. For CPTC, the `ap` and `x` arrays do not overlap and we can disambiguate by declaring `double* restrict ap` and `double* restrict x`. When compiled with the flags `ecl -O3 -Qrestrict`, the assembly code shown in the lower parts is produced. Loop 1 is software pipelined. The software pipeline has 9 stages and the initiation interval (II) is 2 cycles. The latency of a load (`ldfd`) is 9 cycles and the latency of the `fma` that feeds a store (`stfd`) is 8 cycles. The

```
172 for (i__ = j + 1; i__ <= i__2; ++i__) {
173   x[i__] -= temp * ap[k];
174   ++k;
175 }

.b1_76:
 {   .mmi
  (p16) ldfd    f32=[r8],16        //0:173
  (p16) ldfd    f37=[r33]          //0:173
        nop.i   0 ;;
 } { .mmf
  (p16) ldfd    f40=[r3],16        //1:173
  (p16) ldfd    f43=[r38]          //1:173
  (p18) fnma.d  f46=f7,f34,f39 ;;  //9:173
 } { .mmf
  (p20) stfd    [r37]=f48          //18:173
  (p20) stfd    [r42]=f36          //18:173
  (p18) fnma.d  f34=f7,f42,f45 ;;  //10:173
 } { .mii
  (p16) add     r32=16,r33         //3:172
  (p16) add     r37=16,r38         //3:172
  (p16) add     r42=32,r44         //3:172
 } { .mfb
  (p16) lfetch.excl.nt1 [r44]      //3:172
        nop.f   0
        br.ctop.sptk    .b1_76 ;;  //3:172
 }
```

Figure 3: Loop 2. A similar loop with positive increment is also software pipelined and prefetched, but here the compiler chose to unroll the loop by two.

```
214 for (i__ = 1; i__ <= i__2; ++i__) {
215   temp -= ap[k] * x[i__];
216   ++k;
217 }

.b1_107:
 {   .mmi
  (p16) ldfd    f35=[r18],16       //0:215
  (p16) ldfd    f36=[r17],16       //0:215
        nop.i   0 ;;
 } { .mmi
  (p16) ldfd    f32=[r16],16       //1:215
  (p16) ldfd    f37=[r15],16       //1:215
        nop.i   0 ;;
 } { .mii
  (p16) lfetch.excl.nt1 [r34]      //2:214
        nop.i   0
        nop.i   0 ;;
 } { .mfi
        nop.m   0
  (p17) fnma.d  f39=f33,f38,f34    //14:215
        nop.i   0 ;;
 } { .mfb
  (p16) add     r32=32,r34         //9:214
  (p16) fnma.d  f33=f35,f36,f39    //9:215
        br.ctop.sptk    .b1_107 ;;  //9:214
 }
```

Figure 4: Loop 3.

loads are on cycle 0 in stage 1, the `fma` on cycle 9 in stage 5, and the store on cycle 17 in stage 9. Each stage of the SWP is II (=2) cycles long. Notice that stages 2, 3, 4 , 6, 7, 8 of the SWP are empty (the predicates `p17, p18, p19, p21, p22, p23` are absent). They are used to fill the gap between the small II and the high latencies for the load/stores and the `fma`. As we mentioned earlier, for 10-noded tetrahedra, the triangular solves are done on 30×30 matrices. For such a matrix, loop 1 has an average trip count of 15. With a spin up and down of 9 stages, the SWP overhead is considerable.

Loop 2 is better optimized. The assembly output is shown in the lower part of Figure 3. The loop is software pipelined, there are 5 stages in the SWP, and the II is 4. Stages 2 and 4 are empty. Furthermore, the compiler does unroll the loop by two. Why doesn't the compiler unroll loop 1? The only qualitative difference between loops 1 and 2 is that loop 1 is executed in a decremental, rather than an incremental fashion. A brief inspection of loop 1 shows that it can be rewritten with a positive increment. The loop is then almost identical to loop 2 and will be unrolled by two by the compiler. The resulting assembly code is identical to the one shown in the lower part in Figure 3.

5.1.2 Loops 3 and 4

Loop 3, as shown in the upper part of Figure 4 is software pipelined when compiled with `ecl -O2` (II of 5, 2 stages). The assembly code shown in the lower part is produced when compiled with `ecl -O3 -Qrestrict`. It is software pipelined (II of 10, 2 stages) and unrolled by two.

With loop 4, not shown here, we face the same problem as with loop 1. It is almost identical to loop 3, its SWP is a little top-heavy, and it is not unrolled because the compiler does not like decremental loops. The fix for loop 1, rewriting as incremental loop, does the job in this case as well.

Two remarks

For CPTC, the MKL version of `DTPSV` turned out to be slower than our hand-compiled version.

As we pointed out in Section 2.2, `DTPSV` is invoked twice per call to `DPPTRS`. Since `DPPTRS` is invoked very often (once for each element in each CG iteration!), overheads in the SWP might considerably degrade performance. The average workload in `DTPSV` is still relatively small when compared to the spin up and down costs of the SWP. These loops have a relatively small trip count and fairly long SWPs. To make things worse, there are expensive divisions (reciprocals) in the outer loops. An algorithmic change seems to be appropriate. We wrote two functions, `DPPTRS2` and `DTPSV2`, with the same functionality as `DPPTRS` and `DTPSV` except that they are capable of processing two matrices and right hand sides simultaneously. This way we increase the ILP, decrease the number of stages (hence decrease overhead), and increase the number of parallel requests on the front side bus (FSB). (The CPTC performance improved by almost 20%.) From a software perspective, this is a departure from the standard BLAS/LAPACK that generally is not advisable. A more aggressive approach would be to abandon the use of BLAS/LAPACK and rewrite EBE so that the backsolves are done in a "super-loop" for all elements at the same time.

5.2 Sparse Matrix-Vector Multiplication

The CPTC code uses PETSc's CG implementation. PETSc's default storage format for sparse matrices is compressed sparse row format (CSR). Although PETSc has support for blocked formats (block compressed row and block diagonal storage), the FEM formulation in the current version of CPTC is done in a way that destroys the block structure (condensation of essential boundary conditions) and prevents us from using a blocked format. Nevertheless, a

```
571 do j=1,lastrow-firstrow+1
572    sum = 0.d0
CCC assembly starts here
573    do k=rowstr(j),rowstr(j+1)-1
574       sum = sum + a(k)*p(colidx(k))
575    enddo
576   w(j) = sum
577 enddo
```

Figure 5: Sparse matrix-vector product from NAS CG benchmark.

```
.b2_15:
 {   .mmi
  (p16) ld4     r40=[r9],4          //0:574
  (p16) ldfd    f32=[r8],8          //0:574
  (p17) shladd  r33=r2,3,r30 ;;     //3:574
 } { .mmi
  (p17) add     r65=-8,r33          //4:574
  (p16) lfetch.excl     [r28],4     //1:573
        nop.i   0 ;;
 } { .mfi
  (p17) ldfd    f37=[r65]           //5:574
  (p20) fma.d   f41=f36,f40,f43     //14:574
  (p16) sxt4    r2=r40              //2:574
 } { .mib
  (p16) lfetch.excl.nt1 [r27],8     //2:573
        nop.i   0
        br.ctop.sptk    .b2_15 ;;   //2:573
 }
```

Figure 6: Sparse matrix-vector product from NAS CG benchmark (assembler code).

certain structure, in the form of nonzeroes in identical column positions in consecutive rows, is preserved. This kind of structure is supported in PETSc by *i-nodes*. PETSc scans the matrix for sufficiently many i-nodes and automatically chooses optimized (for i-nodes) versions of matrix operations. For CPTC problems the natural i-node size is three (three displacements per FE node). We simplify our discussion of the sparse matrix-vector product by considering the "i-node free" version. All the optimizations that we discuss in the following carry over to the i-noded case, the latter actually being simpler than the case we are about to discuss (because there's more work in the loop body and better spatial locality).

Figure 5 shows the sparse matrix-vector multiplication from the NAS CG benchmark [8].

The assembly output (`efl -O3`) for the core loop is shown in Figure 6. It is software pipelined (5 stages, stages 3 and 4 empty, II of 3) and prefetched (stride-one accesses). On our test system, for the class A problem (the matrix A is a $14,000 \times 14,000$ matrix and 15 iterations), the performance for the entire CG algorithm is 88 MFlops, which is poor when compared to our conservative estimate of 250 Mflops in Section 3.2. There are two basic issues:

1. There is not much work done in line 574, which can be helped by unrolling.
2. The array `p` is not automatically prefetched.

Figure 7 shows an optimized version of the matrix vector product. The loop was unrolled by two, the loads are

```
579 do j=1,lastrow-firstrow+1
580   i = rowstr(j)
581   iresidue = mod( rowstr(j+1)-i, 2 )
582   sum0 = 0.d0
583   sum1 = 0.d0
584   if( iresidue .eq. 1 ) then
585      sum0 = sum0 + a(i)*p(colidx(i))
586   endif
587   low = i+iresidue
588   high = rowstr(j+1)-2
CCC assembly starts here
589   do k=low,high,2
590      i0 = colidx(k)
591      a0 = a(k)
592      i1 = colidx(k+1)
593      a1 = a(k+1)
594      p0 = p(i0)
595      p1 = p(i1)
596      call lfetch_nt1(p(colidx(k+8)))
597      call lfetch_nt1(p(colidx(k+9)))
598      sum0 = sum0 + a0*p0
599      sum1 = sum1 + a1*p1
600   enddo
601   w(j) = sum0 + sum1
602 enddo
```

Figure 7: Optimized version of the matrix vector product with manual prefetching and unrolling by 2.

hoisted, and the array `p` was manually prefetched. The assembly output (`efl -O3`) for the core loop is shown in Figure 8. It is software pipelined (2 stages, II of 8 cycles) and the pipeline is much shorter this time. On our test system, for the class A problem, the performance for the entire CG algorithm is 187 MFlops.

PETSc's i-noded version of sparse matrix-vector multiplication is already unrolled. It only needs to be manually prefetched (5 prefetches for i-node size 3). The file to be modified is `$PETSC_DIR/src/mat/impls/aij/seq/aijnode.c` and the function `MatMult_SeqAIJ_Inode` needs to be modified.

5.3 The Gather/Scatter

In Figure 9 the core loop of the EBE preconditioner is shown. When compiled with `ecl -O3`, an inspection of the assembly code revealed that none of the loops was software pipelined.

A closer examination raises the following issues:

1. The accesses to the C++ vector `z` in lines 14 and 33, through the corresponding read and write access operators `operator[](size_t)` are declared as `inline`, are treated as function calls, and that is the main reason why these loops are not software pipelined.
2. The expressions in lines 14 and 32 should be simplified, and the pointer indirections should be eliminated.
3. The array `iebe->S` can be aliased and precalculated.
4. The accesses to `iebe->x_locp` in line 15 and the access to `iebe->y_locp` in line 32 are not stride-one, and the compiler will not prefetch them. (Arrays with stride-one access are automatically prefetched by the compiler. This is currently not done for arrays with indirect indices. Intel's compiler group is working to add this optimization.)

```
.b2_17:
{   .mmi
  (p16) ld4     r35=[r10],8        //0:592
  (p16) lfetch.excl     [r34]      //0:589
        nop.i   0 ;;
} { .mmi
  (p16) ldfd    f32=[r9],16        //1:591
  (p16) ld4     r36=[r8],8         //1:590
        nop.i   0 ;;
} { .mmi
  (p16) ld4     r38=[r3],8         //2:596
  (p16) ld4     r39=[r2],8         //2:597
  (p16) sxt4    r40=r35 ;;         //2:595
} { .mii
  (p17) lfetch.nt1      [r37]      //11:597
  (p16) sxt4    r35=r36            //3:594
  (p16) shladd  r41=r40,3,r30      //3:595
} { .mmi
  (p16) ldfd    f34=[r29],16 ;;    //3:593
  (p16) add     r37=-8,r41         //4:595
  (p16) sxt4    r41=r39            //4:597
} { .mii
  (p16) shladd  r36=r35,3,r30      //4:594
  (p16) sxt4    r40=r38 ;;         //4:596
  (p16) shladd  r38=r40,3,r30      //5:596
} { .mmi
  (p16) add     r35=-8,r36         //5:594
  (p16) ldfd    f37=[r37]          //5:595
  (p16) shladd  r39=r41,3,r30 ;;   //5:597
} { .mfi
  (p16) lfetch.excl.nt1 [r27],16   //6:589
  (p17) fma.d   f41=f35,f38,f42    //14:599
  (p16) add     r37=-8,r38         //6:596
} { .mfi
  (p16) ldfd    f39=[r35]          //6:594
        nop.f   0
  (p16) add     r36=-8,r39 ;;      //6:597
} { .mfi
  (p16) lfetch.nt1      [r37]      //7:596
  (p17) fma.d   f35=f33,f40,f36    //15:598
  (p16) add     r32=16,r34         //7:589
} { .mib
        nop.m   0
        nop.i   0
        br.ctop.sptk    .b2_17 ;;  //7:589
}
```

Figure 8: Optimized version of the matrix vector product with manual prefetching and unrolling by 2 (assembler code).

```
 1  for (int i=0 ; i<M.numberOfElements() ; ++i)
 2  {
 3    int ndof = M[i].degreesOfFreedom(), k, offset;
 4
 5    Vector  z(ndof) ;
 6
 7    //----------------------------------------
 8    // Gather and apply S^{1/2}
 9    //----------------------------------------
10
11    offset = iebe->ele_offset[i];
12    for (k=0 ; k<ndof ; ++k)
13    {
14      z[k] = iebe->S[offset]
15       *iebe->x_locp[iebe->ele_dof_to_loc[offset]];
16      ++offset;
17    }
18
19    //----------------------------------------
20    // Triangular solves
21    //----------------------------------------
22
23    M[i].solve(z);
24
25    //----------------------------------------
26    // Apply S^{1/2} and scatter
27    //----------------------------------------
28
29    offset = iebe->ele_offset[i];
30    for (k=0 ; k<ndof ; ++k)
31    {
32      iebe->y_locp[iebe->ele_dof_to_loc[offset]]
33               += iebe->S[offset]*z[k];
34      ++offset;
35    }
36  }
```

Figure 9: EBE core routine.

In Figure 10, an optimized version of the core loop of the EBE preconditioner is shown. The following changes were made:

1. We removed all C++ syntactic sugar. The compiler cannot handle it, for now (as of version 5.0.1B-30.2). We introduced the static `z` array in line 6 and invoke the triangular solve directly in line 44.
2. The expressions in lines 14 and 32 were simplified and the pointer indirections were eliminated.
3. We introduced the array `S` to alias and precalulate `iebe->S`.
4. Using intrinsics (lines 1 and 2), the arrays `iebe->x_locp` and `iebe->y_locp` are manually prefetched.
5. Alignment to 16 byte boundaries for automatic variables is not guaranteed by the compiler and is enforced in lines 6 and 7. It also enables the compiler to issue a load-pair instruction (`ldfpd`) in line 52.

All loops (lines 27, 33, and 50) are now software pipelined.

6. SUMMARY OF RESULTS

We achieved a 3.42-fold speedup in the time per solver iteration, reducing the execution time from 0.486 seconds to 0.142 seconds. Table 3 shows an extract from our "CPTC logbook." It reflects the actual optimization progress. In the following, we look at the optimizations from a different

Table 3: Extract from CPTC logbook. Note that the only algorithmic change was the creation of the routine `DTPSV2`.

Change	Time (s)	Speedup
Baseline	0.486	1.0
SWPipeline all relevant loops in `DTPSV`	0.340	1.43
No C++ exception handling and aliasing of pointers in EBE	0.244	1.99
SWP and prefetch gather/scatter	0.237	2.05
Remove C++ syntactic sugar from EBE	0.217	2.24
Alignment in EBE and load pair	0.207	2.35
MKL 5.0 BLAS	0.247	1.97
Hoist loads and prefetch (16 ahead) sparse matrix-vector product	0.186	2.61
`DTPSV2`	0.155	3.14
Better prefetch in EBE	0.142	3.42

```
#include <ia64intrin.h>
#include <xmmintrin.h>

...

__declspec(align(16)) double z[30];
__declspec(align(16)) double S[30];
__declspec(align(16)) int    loc[30];

for (int i=0 ; i<M.numberOfElements() ; ++i)
{
  int ndof = iebe->ndof[i];
  int k, offset, info, nrhs = 1, *loc_ptr,
      n = ndof;
  char *uplo = "L";
  double xloc, yloc, *S_ptr, *x_locp_ptr,
         *y_locp_ptr;

  //----------------------------------------
  // Gather and apply S^{1/2}
  //----------------------------------------

  offset     = iebe->ele_offset[i];
  S_ptr      = &(iebe->S[offset]);
  loc_ptr    = &(iebe->ele_dof_to_loc[offset]);
  x_locp_ptr = iebe->x_locp;
  y_locp_ptr = iebe->y_locp;

  for (k=0 ; k<ndof ; ++k)
  {
    loc[k] = *loc_ptr++;
    S[k] = *S_ptr++;
    __lfetch(_MM_HINT_NT1, x_loc_ptr+loc[k]);
  }
  for (k=0 ; k<ndof ; ++k)
  {
    xloc = *(x_locp_ptr + loc[k]);
    z[k] = S[k] * xloc;
    __lfetch(_MM_HINT_NT1, y_loc_ptr+loc[k]);
  }

  //----------------------------------------
  // Backsolve
  //----------------------------------------

  cptc_dpptrs(uplo, &n, &nrhs, iebe->factor[i],
              z, &n, &info);

  //----------------------------------------
  // Apply S^{1/2} and Scatter
  //----------------------------------------

  for (k=0 ; k<ndof ; ++k)
  {
    yloc = S[k] * z[k];
    *(y_locp_ptr + loc[k]) += yloc;
  }
}
```

Figure 10: Optimized EBE core routine.

angle and summarize the lessons we have learned along the way.

6.1 CPTC Improvements

The Itanium processor comes with extensive performance monitoring capabilities. The architectural and microarchitectural events on the Itanium processor whose occurrences are countable through performance monitoring mechanisms are well documented [1, 6].

Table 4 shows performance monitoring results for CPTC and compares the baseline and the final versions of the code. The results were obtained using the powerful EMON tool developed by Intel. Unfortunately, EMON is not available publicly. However, similar utilities are available from SGI and HP.

The second row in Table 4 is based on the `DATA_ACCESS_CYCLE` counter which counts the number of cycles that the pipeline is stalled or flushed due to instructions waiting for data on cache misses, L1D way mispredictions, and DTC misses. "Data access cycles" is the ratio between this counter and the total cycles.

The significant decrease in clocks per instruction indicates a greater ILP which is confirmed by a higher average of instructions between stop bits. There is a remarkable decrease in data access cycles resulting in higher throughput. There is also a noticeable shift from the 16-32 towards the 8-16 clock latency range in the cache misses. Finally, our bandwidth utilization almost quadrupled. However, it is still only about 50% of what STREAM told us would be available. The Mflops rate appears reasonable in the light of our crude estimate in Section 3.2.

The VTune Performance Analyzer's output of the execution time distribution for the initial and final versions is shown in Table 5.

6.2 Lessons Learned

The compiler technology for the Itanium architecture is relatively new and has not yet (and cannot be expected to have) reached the level of maturity the technology has for the Pentium architecture.

Considerable performance gains can be achieved when stan-

Table 4: Selected performance counters.

	Baseline	Final
Clocks per instruction	1.56	0.75
Data acess cycles	0.87	0.31
Average intructions between all stop bits	3.2	4.8
Misses with latency between 4-8 clocks	80.1%	80.3%
Misses with latency between 8-16 clocks	11.7%	14.6%
Misses with latency between 16-32 clocks	5.9%	2.0%
Misses with latency between 32-64 clocks	1.0%	0.9%
Bus bandwidth	129 MB/s	506 MB/s
Mflops	57	207

Table 5: Distribution of execution time according to VTune Performance Analyzer.

Function	Baseline	Final
`DTPSV`	47.0%	8.5%
`DTPSV2`	N/A	35.1%
Sparse Matrix-Vector Product	30%	23.6 %
Gather/Scatter	8%	20.7%

dard optimization techniques, supported by a diagnostic tool like the VTune Performance Analyzer, are applied.

For "loopy" codes the following remarks apply.

1. Pay special attention to expression simplification, aliasing and disambiguation.
2. Analyze load/store dependencies and other issues that could make the compiler prefer an overly conservative resolution for potential ambiguities or false dependencies.
3. Be aware that the compiler (in level `O3`) will prefetch arrays only for stride-one accesses. Significant performance gains are possible through manual prefetching with the `__lfetch(*,*)` intrinsics.
4. Examine the assembly output and estimate the number of cycles needed for the loop using as many functional units of the CPU as possible. Check the number of loads, stores, and prefetches in the (C or Fortran) source code, divide it by two and compare it to the length of the loop scheduled by the compiler.
5. Examine the SWP output for your core loops carefully. Questions to ask:
   - Why does the compiler fail to software pipeline a loop?
   - What is the loop's trip count?
   - What is the ratio between the workload in the loop and the overhead introduced by SWP? Loops with low trip counts and many stages in the SWP should be avoided. Can I squeeze more work into the loop's body?
   - Why didn't the compiler unroll this loop? Can I help it by unrolling manually?
6. Check the alignment of your data and use, if necessary, the `__declspec(align(*))` primitive. Cache line splits and expensive misaligned accesses are the potential problems. Also, the compiler will not issue load pair instructions for misaligned data.

6.3 A Few Remarks on Pentium Architecture

This paper is not about comparing Pentium and Itanium architectures. Without additional "boundary conditions" in form of a specific application, cost, amount of porting/tuning involved, any high-level comparison is a fairly futile undertaking. The two architectures are as different as they probably can be, and they are also targeted at completely different audiences. *If the obscurity of the Pentium architecture may appeal to a few people, the Itanium architecture certainly is a contemporary computer-architectural EPIC.*

After folding the changes for the Itanium processor back into the CPTC source tree, the performance improved only marginally on various Pentium-based systems. This can be partly attributed to the maturity of compilers for the Pentium line. On the other hand, for CPTC, vectorization alone will not yield dramatic performance gains and SWP is clearly the more flexible and powerful technology. Furthermore, we believe in the greater advantage of EPIC over runtime ILP detection mechanisms.

The performance of the CPTC on the Itanium processor, as the first implementation of EPIC, is certainly competitive with that of current generation Pentium 4 systems.

7. CONCLUSIONS

Although the Itanium processor is the first implementation of the Itanium architecture, it is a great processor for scientific computing. The Itanium architecture offers tremendous resources in the form of numerous functional units and a wealth of general purpose and floating point registers. We have found it advantageous that the compiler exercises full control of the utilization of those resources. The job of optimization becomes that of enabling the compiler to exploit parallelism in the code and optimize utilization of the memory subsystem. Compilers for the Itanium architecture are maturing, but, with significant programmer help, achieve top performance. Future performance increases will be available with Itanium architecture follow-ons (McKinley) and compiler improvements.

8. ACKNOWLEDGMENTS

We would like to thank Intel® Solution Services for their hospitality at their Intel® Solution Center (formerly known as Intel Application Solution Center) in Chandler, AZ. Intel Solution Centers are state-of-the-art facilities for designing and testing high performance solutions. During two inspiring and extremely productive weeks of collaboration, we were able to test, tune and optimize solutions on Intel technology.

The first author was supported by the National Science Foundation's grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.

The second author was supported by the National Science Foundation's grants KDI-9873214 and EIA-9972853.

We would like to thank the head of the Cornell Fracture Group, Prof. Anthony R. Ingraffea, for his interest in and support of this project.

We gratefully acknowledge the support of the Dell Computer and Intel Corporations for providing Itanium processor-based hardware to port the CPTC environment at the Cornell Theory Center, and the support of the Intel and Microsoft Corporations for donating the necessary software.

9. REFERENCES

[1] Intel IA-64 Architecture Software Developer's Manual, 2000. Volumes 1–4, Revision 1.1.

[2] CLAPACK @ Netlib. http://www.netlib.org/clapack/index.html, 2001.

[3] Cornell Fracture Group home page. http://www.cfg.cornell.edu/, 2001.

[4] CPTC home page. http://www.tc.cornell.edu/Research/CMI/CrackProp/index.asp, 2001.

[5] F2C @ Netlib. http://ftp.netlib.org/f2c/index.html, 2001.

[6] Intel Itanium Architecture Software Developers Manual Specification update. http://developer.intel.com/design/itanium/manuals/248699.htm, 2001.

[7] MPI home page. http://www.mcs.anl.gov/mpi/, 2001.

[8] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/, 2001.

[9] OpenMP home page. http://www.openmp.org/, 2001.

[10] STREAM home page. http://www.cs.virginia.edu/stream/, 2001.

[11] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1994.

[12] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, L. C. McInnes, and B. F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc/, 2001.

[13] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[14] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.0, Argonne National Laboratory, 2001.

[15] E. A. et al. *LAPACK User's Guide*. SIAM, 1999.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 1996.

[17] I. Hladik, M. Reed, and G. Swoboda. Robust Preconditioners for Linear Elasticity FEM Analyses. 40:2109–2117, 1997.

[18] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

Dynamic Binary Instrumentation on IA-64

Vinodha Ramasamy
Hewlett Packard Company
19111 Pruneridge Avenue, ms42u4
Cupertino, CA - 95014
1-408-447-2819
vinodha@cup.hp.com

Robert Hundt
Hewlett Packard Company
19111 Pruneridge Avenue, ms42u4
Cupertino, CA - 95014
1-408-447-0546
rhundt@cup.hp.com

ABSTRACT
Dynamic binary instrumentation is the process of modifying the instructions of a binary program while it executes. This technique is used in a wide variety of software engineering domains such as performance analysis, program optimization and quality assurance. HP Caliper provides a framework for building such tools and combines function-level dynamic instrumentation with other measurement techniques. This paper describes in detail the dynamic instrumentation framework provided by HP Caliper. It discusses the specific constraints imposed by the EPIC architecture and the challenges faced in making this a full-fledged, industrial-strength product handling a wide range of real-world applications.

General Terms
Algorithms, Measurement, Performance.

Keywords
Dynamic Binary Instrumentation, HP Caliper, IA-64.

1. INTRODUCTION
Binary instrumentation is the technique of modifying a binary program. Instructions are added, modified or deleted. Static and dynamic data such as relocation information or procedure lookup tables may also be rewritten.

Modifications may be applied to the program's executable file, object files, or class files and at any phase - compile time, link time, load time or run-time. Instrumentation may be applied to complete load modules, functions, super blocks, basic blocks or individual instructions. Systems performing instrumentation include performance analysis tools [2][11], software simulators, binary translators [3][34], virtual machines (such as the Java VM) and even debuggers [8][10].

This paper focuses on function level dynamic instrumentation as implemented in HP Caliper [19], a framework for performance analysis tools on IA-64 HP-UX systems. We believe that the techniques we developed and the problems we solved are general and can be applied to many other instrumentation and code generation systems.

EPIC1 Workshop, MICRO34, Dec 1-5, 2001, Austin, Texas.

What are the characteristics of dynamic binary instrumentation?

Dynamic instrumentation operates on a running process. By using dynamic code injection techniques, no special preparation or recompilation of the executable is necessary, since the instrumentation code is generated during the execution of the application. Moreover, only the portion of the code that is executed at runtime needs to be instrumented. Experiments show that typically only a small fraction of the entire code is reached during a single program run (validating the "80:20 rule"). This makes dynamic instrumentation scalable in terms of memory usage and performance when dealing with large applications.

It also becomes possible to employ "lazy mechanisms" to generate auxiliary data only when needed. For example, the creation of unwind data to be used for unwinding stack through instrumented functions can be postponed until an actual stack unwind operation occurs. Varying instrumentations can be applied during different runs of the same application, for example to collect different types of profile data or to augment an existing profile database.

What are the challenges for such a dynamic instrumentation system?

Dynamic instrumentation, by its very nature, inherits the runtime behaviour of the application being instrumented. The exact nature of the problems to be solved by a dynamic instrumentation system will therefore depend not only on the specific algorithmic choices for the instrumentation strategy, but also on the types of applications to be supported, the constraints specific to the operating system, and the architectural features of the underlying hardware on which the framework is executed.

What are the IA-64 architectural features that impose additional challenges?

The IA-64 is a very large instruction word (VLIW) architecture in which multiple instructions are grouped into bundles. Thread-safe code patching of bundles becomes an issue.

A call branch instruction may be predicated and followed by other instructions in the same bundle. If the call branch is taken, the other instructions in the bundle will not be executed. This introduces the notion of call shadow instructions which need special handling by the instrumenter.

The use of branch registers for returning control to the caller function complicates the mechanism required to clean up the call stack when instrumentation is to be undone.

To implement support for control speculation in the IA-64 hardware, each general register is associated with a "not a thing" (NaT) bit to track deferred speculative exceptions. The instrumentation code will need to save and restore a general register along with its associated NaT bit in order to preserve the original

program semantics. The exact nature of these problems and their solutions are discussed in detail in later sections of this paper.

Why do dynamic instrumentation at all?

The following are compelling reasons for performing dynamic instrumentation: The ability to see the entire runtime context, no special preparation of the application to be measured, scalability and speed.

Most of the current generation of applications are complex networks of inter-connected components that don't exist in source or object form at the same time, and therefore cannot be seen in full context by static instrumentation tools. A dynamic tool has the opportunity to see the entire context of an application. For example, the dynamic model can track the loading and unloading of shared libraries or Dynamic Load Libraries (DLLs) and therefore support their "automatic" instrumentation.

Dynamic instrumentation is widely used to build performance analysis and coverage tools that can operate on the applications to be measured without the need for a special compile or link process. They are also used to gather profile data that can be used by profile-guided compilers to optimize the code generated based on the dynamic or run-time behaviour of the application. Such performance analysis tools become even more important for the EPIC architecture, where several architectural features such as speculation, predication and branch prediction explicitly create synergy between the compiler and the processor.

Additionally, with its wide address space, the EPIC architecture fosters the development of huge applications. Tools must therefore scale well to large application sizes.

The rest of the paper is organized as follows. First, we describe the core components and framework of a dynamic instrumentation system (DIS). Then we outline the (dynamic) instrumentation of a single function. We address algorithmic choices, compiler annotations, switch tables, free registers, indirect branch targets, issues related to branch ranges, instruction pointer sensitive code, and call shadows.

Next, we discuss the challenges imposed by the runtime architecture. In this context we explain application specific challenges, such as multi-threaded programs, programs that create new processes, C++ applications that throw exceptions or otherwise unwind the stack, and uninstrumentation in general.

We conclude with the description of an application of dynamic instrumentation for profile based compiler optimization (PBO). We describe the specific instrumentation applied for this purpose and analyze various aspects of its performance overhead.

For a complete understanding of the presented material it has been necessary to reference parts of [20] and [22]. These references however have been improved and updated.

2. INSTRUMENTATION SYSTEM

Dynamic instrumentation operates on a running process. A support system that manages processes and the instrumentation is therefore needed. We call such a system a dynamic instrumentation system (DIS). For such a system there are many design choices to be made:

- Method to start or attach to a process.
- Granularity of instrumentation - for example, the instrumentation could be applied to whole load modules, functions or individual instructions.
- Location and execution of the initialization and support code, as well as of the instrumented code.
- Control transfers between the instrumented application and the DIS.
- Memory allocation for probe code sequences - for example, via shared memory or via in-process malloc() calls.
- Storage for measurement data - for example, the data manipulated by the probe code can be in shared memory, or stored in another process' address space using interprocess communication (IPC) mechanisms.

This section outlines the dynamic instrumentation system as implemented in HP Caliper. In the following descriptions, the term target refers to the application process being profiled.

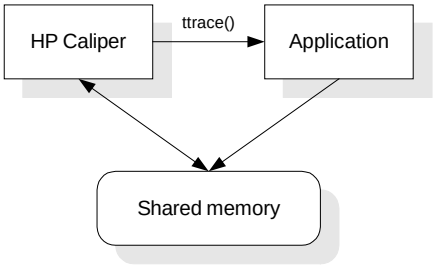


Figure 1: Dynamic Instrumentation System HP Caliper

HP Caliper controls a target application through the operating system's debug interface (ttrace() on HP-UX) and achieves control transfers to and from the application with the help of parameterized breakpoints (Figure 1). The breakpoint parameters help in identifying the appropriate action to be taken by the HP Caliper process for the current control transfer.

Our dynamic instrumentation algorithm consists of the following five steps as displayed in Figure 2.

1. *Attach and Inject*: HP Caliper identifies an executable or an already running target process and attaches to it using the HP-UX *ttrace()* system call. The target process stops and transfers control to HP Caliper, which then maps shared memory into the target process' address space using a dynamic code injection technique. This shared memory space is accessible by both the HP Caliper and the target process and will hold the instrumented code and data. This enables HP Caliper to instrument executables without the need for any special compilation or modification to the build/link process.

2. *Function Discovery*: All the entry points of functions are identified by reading the symbol table entries present in an executable binary file. Functions may have multiple entry points, for example, the C++ constructors and destructors as implemented by the HP compilers. Functions may also have been split into hot and cold regions. It is also possible to dynamically detect function entry points as they are reached during the program's execution.

3. *Static Break Insertion*: Every function's entry points are patched with a break instruction. It is possible to exclude functions from instrumentation by not placing breakpoints at their entry points.

4. *Run under Dynamic Instrumentation:* Control is transferred back to the process. The process runs until it hits one of the inserted break instructions. Since the process is controlled by ttrace(), control transfers to HP Caliper. The reached function is instrumented and placed in shared memory. The original function's entry point is patched with a long branch instruction to its instrumented version.

When possible, call sites in other instrumented functions that call the current function are patched to directly call the new (instrumented) version. Calls to the original function that weren't patched will still be transferred to the instrumented version by the long branch instruction.

Execution of the target is resumed at the instrumented function's entry point. The target continues execution until it hits another breakpoint at an uninstrumented function's entry. Control again transfers to HP Caliper and the dynamic instrumentation process is repeated.

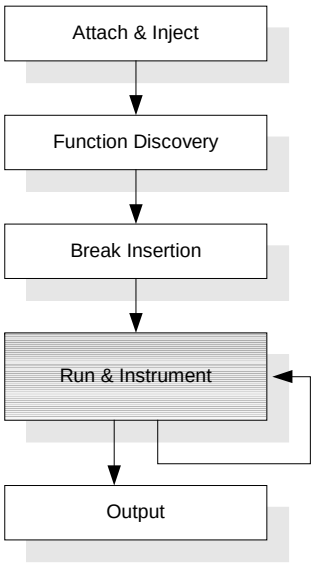


Figure 2: The dynamic instrumentation algorithm

5. *Output*: Upon process termination or user request, control again transfers to HP Caliper. Statistics, counters, and other measurement results are now retrieved and presented.

2.1 Design Alternatives

Rather than have HP Caliper operate as a separate process, a design alternative is to inject the HP Caliper code into the application process's address space at application startup time. This would require modification to the kernel in the process startup sequence, as done to support Aries [34]. However, this approach will not allow HP Caliper to attach to an already running program, and was therefore not chosen.

Writing into shared memory by the HP Caliper process is much faster than using ttrace() to write into the measured application's data space. Hence we decided to use shared memory to hold the instrumented code and data that will be accessed by both the HP Caliper process and the application process as opposed to allocating memory in the application's address space.

2.2 Core Components

The core components of HP Caliper's DIS and their key functionality are now described.

Binary reader - The binary, which is in the "Executable and Link Format (ELF)" on HP-UX, contains the symbol table information that is used to identify function boundaries, text sections with the original code, compiler annotations, and other program data that is used by the instrumenter. The binary reader provides the functionality of obtaining these. It handles both 32 and 64 bit ELF files.

Process handler - This module handles all the process related tasks such as starting or attaching to the process to be instrumented, handling process events such as calls to fork() and exec(), and maintains the load address mapping for shared libraries that are loaded/unloaded. It also provides the code injection mechanism needed at startup to map the shared memory into the target address space.

Function discovery – This component identifies the original function boundaries by reading the symbol table information present in the target binary. It maintains a function dictionary that is the repository of all information regarding the address ranges, sizes and other attributes of both the original functions and the instrumented functions.

Code patcher - The code patcher provides thread-safe capabilities to dynamically patch data and code in the target process. For example, the code patcher is used to insert break instructions at function entry points and to later replace the reached break instructions with branch instructions to the instrumented functions. It is also used to patch data such as switch tables in the target process.

Decoder - The decoder provides the functionality to read in the original function code and decode it into an intermediate representation (IR). The IR is made up of instructions and blocks and allows operations such as insertion, deletion and duplication of instructions and blocks.

Encoder - The encoder is used to convert the IR fragment representing the instrumented function into its corresponding binary bits. The encoded bits are written into shared memory. The encoder also provides a templatizer and scheduler, to pack the instructions into IA-64 bundles.

Instrumenter - Given an original function address and size, the instrumenter decodes the function, inserts probe code and produces an instrumented copy of the function in shared memory. The details of the instrumenter is the focus of Section 3.

Unwind handler – This module manages all aspects of unwind data handling. Unwind information is read, updated and written. Code is generated and injected into the target process to register modified unwind information to the run-time system.

The above components are the core DIS components. Similar components are needed in every (dynamic) instrumentation system. For a specific instrumentation method, additional support code may be needed. For example, in order to support profile based optimization for the HP compilers, the following two components were added.

Counter manager - This module maintains the counters to be used by the inserted probe code. It provides counter sets that can be either statically allocated by the instrumenter or dynamically allocated during the execution of the target (by the probe code inserted into the target).

Profile writer - The measurement results are retrieved from the counters and written into a profile data file that can be fed back to a profile guided compiler.

3. INSTRUMENTATION

Our instrumentation algorithm works at the granularity of functions, i.e., entire functions are instrumented at a time.

3.1 Algorithm

There are several ways to perform function-level instrumentation. We considered the following two strategies:

- Out-of-line instrumentation with the use of trampolines for the probe code
- Inline instrumentation by creating instrumented copies of the function with inlined probe code and relocating the entire function.

Out-of-line instrumentation - This method is most suited when the instrumentation involves simply the insertion of probe instructions without major modifications to the original code, and when the density of the inserted probes is low. Consider the following example of a function entry code:

```
foo::
  bundle1::      alloc  r33 = ar.pfs, 0, 11, 0
                 addl   r9  =  -32, r1
                 addl   r8  =  128, r1
  bundle2::      . . .
```

If a counter is to be inserted at the function entry in order to perform function counting, the out-of-line method will transform the original code as follows (for example):

```
foo::
<bundle1>::  nop.m
             brl    <trampoline>

<bundle2>::  . . .

<trampoline>::
      // Save state (if necessary)
      // to create free register "rx"

      // execute original instructions
         alloc  r33 = ar.pfs, 0, 11, 0
         addl   r9  =  -32, r1
         addl   r8  =  128, r1

      // increment counter
         movl   rx = <counter addr>  ;;
         fetchadd  rx = [rx], 1

      // restore state (if necessary)
         . . .

      // return to original code
         brl  <bundle2>
```

The main advantage of this method is its simplicity – branches in the original code will still continue to reach their targets and therefore no target address translation needs to be done at runtime.

If many probes need to be inserted into one function, then the overhead and code bloat of all these "out to the woods and back" branches is significant. If the trampoline code is not reachable from the original code space using the 25-bit encoded offset of an IP-relative branch instruction on the IA-64, it will necessitate the use of a long branch (brl) instruction for this purpose.

On some of the IA-64 microprocessors (such as Itanium), the brl instruction is emulated with an associated overhead of about 100 to 300 cycles. The overhead associated with this out-of-line approach deemed it infeasible to meet our performance constraints.

Inline method - In the inline method, the probe code is inlined into the original function code to produce an instrumented function, which is then relocated.

For the previous example of function counting, the corresponding instrumented code using this method will be as follows:

```
foo_instrumented::
          // modified alloc instruction
          // to obtain free register
<bundle1>::  alloc  r33 = ar.pfs, 0, 11, 1
             addl   r9  = -32, r1
             addl   r8  = 128, r1

         // update counter – r43 is the
         // (output stacked) free register
             movl r43 = <addr_of _counter> ;;
             fetchadd  r43 = [r43], 1
<bundle2>::  . . .
```

The instrumented copy of the function is relocated (for example, to shared memory). This method leads to more compact code and better performance and was therefore our method of choice. However this choice came with a price – the function relocation made the mechanism of unwinding the call stack for dynamically instrumented programs extremely complicated. The necessity of handling stack unwinding correctly and the type of problems that had to be solved is described in Section 5 in more detail.

3.2 Data Flow

The various steps performed by the frontend, backend and instrumenter are outlined below and illustrated in Figure 3. In this figure, solid arrows represent control flow, while dotted arrows represent data flow.

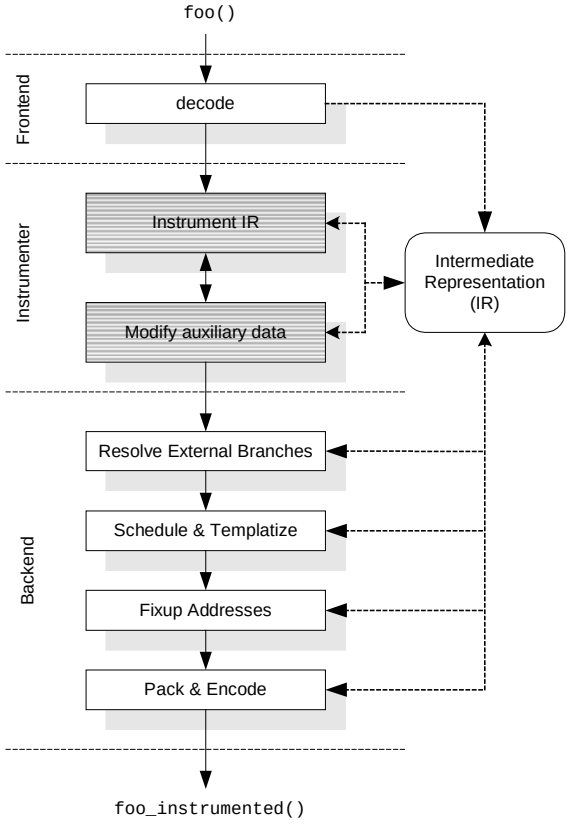


Figure 3: Work breakdown for frontend, instrumenter and backend

The instructions of the original function are read and decoded into an intermediate representation (IR). All further operations to produce the instrumented function, such as insertion, deletion or modifications of the instructions, operate on this IR. Depending

upon the type of measurements to be made, the appropriate probe code is inserted. For example, the probe code could be code to increment counters in order to record execution frequencies, or it could be code to check the values of a function's input parameters for value profiling.

Data such as the original entries of switch tables need to be replaced with their corresponding new targets in the instrumented address space. All ip-relative branches with external targets (i.e., targets that are not within the given function) will have to be fixed. If the target has been instrumented, the branch target is changed to the instrumented target address, otherwise the branch is converted to a long branch instruction to the original target. A list of all branches whose targets have not been instrumented are recorded so that they can be patched to branch directly to the instrumented target, after the target function has been instrumented.

The instructions in the IR are packed into IA-64 bundles. The scheduler can be very complex if an optimal instruction schedule is desired. We, however, use a very simple scheduler that uses a minimal one instruction look-ahead state machine to determine the bundle templates and the instruction schedule.

Internal branch targets can now be fixed since the absolute addresses of all instructions in the intermediate representation is known at this stage.

The code is encoded into IA-64 bundles and written into shared memory. After the instrumented code is written into shared memory, it is necessary to flush this address region from the hardware caches in order to invalidate lines that may have been prefetched into the instruction-only cache (L0I).

If any error condition is detected while instrumenting a function, the appropriate error handling is done. In some error cases, the instrumentation of the function is undone and the break at the function entry is replaced by the original bundle. This ensures that the instrumenter does not try to instrument the function when it is reached again. The user is notified about the functions that were not instrumented through diagnostic messages in the output report.

3.3 Compiler Annotations

The HP-UX compilers provide several annotations in the binary executable as hints for post-compilation tools to avoid detailed control and data flow analysis. Our instrumentation algorithm uses the following two types of compiler annotations:

A first set of annotations is used to identify spare registers in a region of code. These registers can be safely used in the inserted probe code as they are guaranteed to be unused in the original code region.

A second set of annotations is used to locate switch tables. HP-UX compilers implement switch statements in C source code using an indirect (non-call) branch through a table of pc-relative or absolute offsets [1]. The same mechanism is used for C++ catch clauses. These switch tables can be either in the data or the code sections.

The instrumenter needs to locate such switch tables in order to differentiate data (such as the switch table) intermixed in the code section. The switch table entries are branch target addresses in the original uninstrumented function code. When a function is instrumented, these entries need to be replaced with the corresponding target addresses in the instrumented code range.

In the absence of compiler annotations, the instrumenter would need to perform a reverse analysis of the code to identify the location of the switch table by tracking the address moved to the branch register that is used to reference the switch table entry. This analysis can be tedious and error-prone [6][32]. Compiler provided annotations greatly simplify this process.

4. CHALLENGES

This section describes some of the interesting instrumentation problems such as finding free registers to be used in the instrumentation code, thread-safe code-patching, handling of call-shadow instructions and handling of return pointer sensitive code.

4.1 Free Register Allocation

The probe code inserted by the instrumenter needs free registers. For example, the code to increment a counter needs a free register to hold the address of the counter to be incremented. There are several methods used by post-compile tools to obtain free registers. Sometimes these tools use special registers that are reserved for this purpose.

We chose not to use this approach. Our register allocation algorithm tries to obtain registers using the most cost effective methods initially. If these simpler methods fail, it falls back to a method that is more expensive in terms of runtime performance but is guaranteed to provide free registers. The different staged methods to obtain free registers are the following:

Use of compiler annotations - Obtain free registers using the hints provided by the compiler annotation.

Modifying the parameters of an alloc statement - Free registers can be obtained by increasing the number of output stacked registers reserved for the current register stack frame by modifying the parameters of the alloc instruction associated with the code region. The extra stacked registers obtained this way are guaranteed to be unused within the function and hence can be used for the probe code. Moreover, the output stacked registers are not saved by the Register Stack Engine (RSE) across a function call. Therefore this method does not add additional pressure on the RSE.

However, obtaining free registers becomes complicated when there are multiple allocs within a function. In this case, each basic block could be associated with different alloc statements, which would translate into different stacked registers being allocated as free registers for each basic block. The alloc instruction reaching each branch needs to be identified and this analysis can be expensive. So currently this method is used only when there is a single alloc instruction in the function. If the alloc instruction already uses the maximum number of stacked registers (92), then this method will fail to produce spare registers.

Save and restore - This is the fall back method which is guaranteed to provide free registers. The idea is to save the general register (gr) to be used by the probe code onto the memory stack, and restore the gr after the probe code sequence. It is necessary that the NaT bit associated with the gr that is used to track deferred speculative exceptions is also saved and restored correctly to preserve program semantics. Saving and restoring grs along with their associated NaT bit is performed using spill (st.spill) and fill (ld.fill) instructions. However, for the purpose of probe code insertion, this is not possible, since spilling a gr onto the stack on the IA64 modifies the UNAT register. A modified UNAT register may change the program semantics of the original code. Saving and restoring the UNAT register will also require a free register, which we didn't have to begin with!

This problem was solved by the following observation: Moving a gr to and from a floating point (fp) register with the use of setf.sig and

getf.sig instructions preserves the NaT bit of the gr. If the compiler annotations do not yield any free fp registers, then a spare fp register is obtained by saving a scratch fp register onto the memory stack. The stf.spill instruction that is used to save a fp register does not modify the UNAT register and therefore solves this problem.

4.2 Call Shadow Instructions

Call branch (br.call) instructions may be predicated and the state of the predicate controls whether the program control will be transferred to the target address of the branch. Upon completing execution of the code at the target of a taken call branch, control is returned to the instruction bundle that follows the call-branch instruction.

On the IA-64, it is possible that a call branch instruction is scheduled in slot 0 or slot 1 of a bundle, and can therefore be followed by other instructions in the same bundle. These instructions are skipped if the branch is taken, and will be executed if the call branch is not taken. These instructions are referred to as *call-shadow* instructions.

```
{.mbb
            nop.m
      (p6)  br.call rp = <target>
            break.b
}
<next_bundle>::
```

In the above example, if the predicate (p6) is true, the first call branch is executed and control is transferred to the target of the call (<target>). Since the return addresses are always on bundle boundaries, control is transferred to the code at <next_bundle> after returning from the target. The break instruction following the call branch will not be executed in this case.

If probe instructions are inserted in the proximity of the call branch instruction, the bundling of the call-branch and call-shadow instructions may change. I.e., the call-branch may be scheduled in one bundle and the call-shadow instructions in another bundle. For the previous code example this may result in the call shadow instruction being executed even when the predicate (p6) is true. An example scheduling with probe code is shown:

```
{.mmb
            <probe_instruction1>
            <probe_instruction2>
     (p6)   br.call rp = <target>
}
{.bbb
             nop.b
             nop.b
             break.b
}
```

The instrumented code above will be semantically different from the original code, since it changes the implicit logic of the call-shadow.

Therefore, in order to ensure correct instrumentation, code with call-shadow instructions must be identified and handled appropriately. The call-shadow instructions are identified by analyzing the original code for a br.call instruction in slot 0 or slot 1 of a bundle and followed by non-nop instructions in the following slots of the bundle. The original code sequence is then modified to ensure that insertion of probe instructions will not change program behaviour.

Our initial approach was to place a branch around the call-shadow instructions that is predicated by the same predicate as the call branch instruction. However, this assumes that the value of the predicate is not modified by the called target. This will not work if the predicate is a scratch predicate (p6 to p15).

Our next approach was to use trampolines. The instrumented code for the previous call-shadow bundle will be as follows:

```
{.mmb
            <probe_instruction1>
            <probe_instruction2>
     (p6)   br   <trampoline>
}
{.bbb
             nop.b
             nop.b
             break.b
}
<next_bundle>::
             . . .

<trampoline>::
{.mmb
             nop.m
             nop.m
             br.call rp = <target>
}
{.mmb
             nop.m
             nop.m
             br  <next_bundle>
}
```

The templatizer also ensures that no new call shadow instructions are inadvertently introduced. This is achieved by ensuring that call branches are always scheduled in slot 2 of a bundle.

The above solution however introduced complications in unwinding stack through the instrumented code. The data in unwind tables is associated with contiguous regions of code and is used to unwind through instrumented code. The details are explained in Section 5. Our final solution to handle call-shadow instructions (which also handles the stack unwinding problem) was to inline the trampoline code as shown in the following example:

```
{.mmb
            <probe_instruction1>
            <probe_instruction2>
     (p6)   br   <trampoline>
}
{.bbb
            nop.b
            break.b  //original call shadow
            br   <next_bundle>
}
<trampoline>::
{.mmb
            nop.m   // or probe instruction
            nop.m   // or probe instruction
            br.call   <target>
}
<next_bundle>::  …
```

4.3 Handling Indirect Branch Targets

Currently we perform no dynamic handling of indirect branch targets. No lookup is needed to translate indirect call branch targets since their targets are function entry points. There is always either a break at the function entry point for non-instrumented functions, or a long branch to its instrumented counterpart. For handling non-call (internal) indirect branch targets, it is mandatory that all such branches are associated with compiler annotations that identify the possible targets of an internal indirect branch. This is necessary to identify the basic blocks of a function and to ensure correct scheduling (i.e. the target of an indirect branch begins a basic block). If a non-call indirect branch does not have an associated compiler annotation, then the function cannot be instrumented.

4.4 Handling Return-Pointer Sensitive Code

On a function call, the return-pointer (rp), which is a dedicated branch register, is used to return from the called function back to the calling function. Typically the return pointer is set to the address of the bundle following the call. In the code below, rp is set to the address of <bundle2>.

```
<bundle1>::
     <inst1>
     <inst2>
     br.call rp = <target_function>

<bundle2>::
     . . .
```

Some functions may require that the value in the return pointer references an address within the original address space. For example, the function may use the return pointer value to find the load module of the calling function. If the return pointer is an address outside the original function's address range, (as for example, a shared memory address of the instrumented function), it might cause the called function to fail. We call such functions rp-sensitive functions.

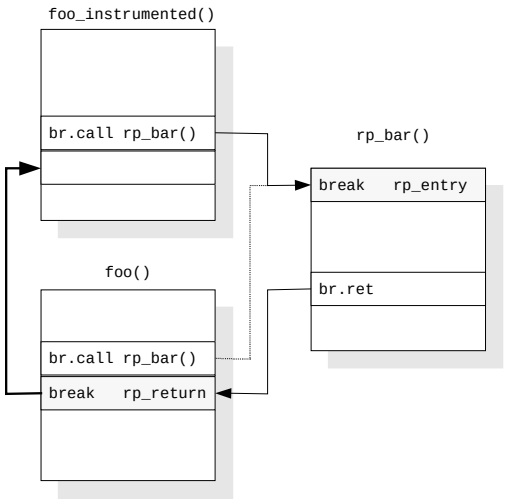


Figure 4: Handling rp-sensitive functions (rp_bar())

To handle this problem, functions that are rp-sensitive are identified and instrumented. A special break instruction, that we will call an rp-entry break, is placed at the instrumented function entry. When such a function is called, the rp-entry breakpoint is hit. This transfers control to the HP Caliper process.

The HP Caliper process now reads the contents of the rp register at this point. If it contains an address in the instrumented address range (rp-instr), the corresponding bundle address in the original function is located (rp-mapped). The mapping <rp-instr, rp-mapped> is recorded and the value rp-mapped is written into the rp register. The bundle in the original non-instrumented function at the address rp-mapped is patched with another special breakpoint, which we will call an rp-return breakpoint. The target process is now resumed at the called rp-sensitive function.

Since the rp register now holds a "valid" address belonging to the original address space, the rp-sensitive function executes as if it was called from the original non-instrumented caller function. On return from this rp-sensitive function, control is transferred to the address rp-mapped using the value in the rp register. Since there is now a break (rp-return) at this address, control is transferred back to HP Caliper. HP Caliper looks up its mappings to locate the address (rp-instr) corresponding to rp-mapped. The value of rp is then reset to its original instrumented caller address (rp-instr) and the target is resumed from this address.

The above mechanism makes the instrumentation transparent to the rp-sensitive functions, and allows us to handle programs with rp-sensitive functions correctly.

4.5 Multi-Threaded Applications

Thread-safe code patching – In order to handle applications with multiple threads, it is necessary to ensure that the dynamic code patching is thread-safe. Our dynamic instrumentation algorithm patches bundles in the original code (for example, it places break bundles at function entry points and later replaces them with brl bundles to branch to the function's instrumented version). The size of an IA-64 bundle is 16 bytes, and the store operations operate at a maximum granularity of 8 bytes. This means that at least two store operations are needed to perform a bundle update. Moreover, a bundle can hold 3 instructions. This leads to the following potentially hazardous scenarios:

1. A thread could hit the bundle in the middle of a patch operation (i.e., after the bundle has been patched partially).
2. A thread could have been stalled in slot 0 or slot 1 of the original bundle, and is restarted on the patched bundle.

One of the code patching sequences that had to be made thread-safe is the writing of brl instructions at the function's original entry point to branch to its corresponding instrumented version. This problem is simplified since the bundle to be patched has a break instruction in slot 0. I.e.,

```
break.m    <constant>
nop.b
nop.b
```

is changed to

```
nop.m
brl   <instrumented_function>
```

The above code patching is made thread-safe by performing the bundle update using 2 stores.

1. First store to update the second half of the bunde
2. Second store to update the first half of the bundle.

The initial placing of break instructions at function entry points performed at function discovery time needs no special handling, if this operation is done before the target has begun execution. In this case, there are no active threads in the target process at the time of break insertion and therefore the sequence in which the bundle is patched does not matter.

Data operations – The probe code inserted into the target process may be used to obtain aggregated data across all threads of the process. This results in the probe code being executed by multiple threads to update data structures that are shared across the threads. The reads and writes to this data need to be made thread-safe. A simple example of such data accesses is the code inserted to increment a shared counter. If this operation is performed using the following sequence of instructions:

```
movl rx = <counter_addr> ;;
ld ry   = [rx]           ;;
add ry  = ry, 1          ;;
st [rx] = ry             ;;
```

it will lead to potential race conditions between the multiple threads performing the store operations resulting in incorrect counts. To avoid such a race conditon, the counter update needs to be

performed using an atomic update operation such as the fetchadd instruction:

```
movl      rx = <counter_addr>  ;;
fetchadd  rx = [rx], 1         ;;
```

The above sequence has the added advantage that it requires only a single free register (rx). Our measurements to compare the runtime overhead between the two sequences showed very similar results, since the scheduler we use is very simple and does not reorder the load/add/store instructions to obtain a better schedule. We therefore chose the more accurate (and convenient) solution of using the fetchadd instead of the ld/add/st sequence.

4.6 Address Handling

Since the instrumented function is relocated, the original targets of chk and ip-rel branches may no longer be reachable. These need to be handled appropriately.

For example, a chk instruction pointing to an address <target> is changed to point to a trampoline, that performs a long branch to the original target.

```
(p1) chk.a  <target>
     [. . .]
```

is changed to

```
(p1) chk.a  <trampoline>
     [. . .]

<trampoline>:: brl <target>
```

Instructions that rely on the value of the original instruction address (ip-relative code) such as

```
mov rx = ip
```

should be replaced with

```
movl rx = <original ip value>
```

Code that calculates or loads a shared memory address as an indirect branch target address will need to explicitly have instructions to swizzle this pointer address to handle 32 bit applications.

5. UNINSTRUMENTATION

In the previous sections, we discussed the mechanisms to instrument a function and execute the instrumented code. In some instances it is desirable or necessary to do the reverse operation, i.e., to undo the instrumentation and revert back to executing the original code. For example, this capability is useful when an application is to be measured for only a part of its total runtime.

Another use of this capability is the following scenario: A process may have been instrumented to collect certain measurements. This process being measured (the parent process) may create new processes (the child processes). The most common way to create a new process is to call the C function fork() from the parent process. It may be required to exclude some (or all) of the child processes from the measurement of the parent process.

For example (illustrated in Figure 5), a Caliper process (lets call it process B) monitors a target process by creating it as its child process. If the performance bottlenecks of the Caliper process (B) need to be identified, we may run another Caliper process (say A) to measure Caliper process (B). In this case the child process created by the Caliper process (B) needs to be excluded from the measurement.

On HP-UX, the child process will inherit the parent's complete context. This includes the parent's program text which may have been modified by instrumentation, the mapped shared memory segments and the parent's return call stack. If no special handling is done at this point, the child process will execute the inherited instrumented code from the parent, thereby perturbing the measurements being made exclusively on the parent process. In order to exclude the child process from any measurements, the child process should inherit the "clean" non-instrumented state of the parent process – i.e., all modifications in the inherited context of the child process need to be undone. We call this step *uninstrumentation.*

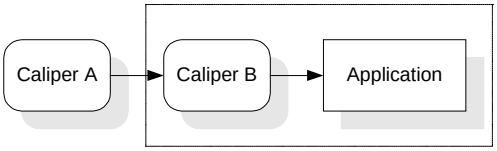


Figure 5: Caliper on Caliper

The shared memory can be unmapped using the same code injection technique that was used to map the shared memory. Control transfers to the instrumented code in shared memory via branches in the text section can be undone by restoring the process's original text segment. Similarly, patched data such as switch tables can be restored to their original state by reading the binary file. If the instrumentation was performed using the out-of-line approach with trampolines, then at the time of uninstrumentation, all threads having their IP within trampolines must have their IP reset to the corresponding non-instrumented address.

However, if the trampolines used in this out-of-line instrumentation method have calls to other instrumented functions, or if in-line instrumentation is used, then the call stack will contain addresses in the instrumented address range. The complexity in the uninstrumentation phase comes from the necessity to *clean up* this call stack.

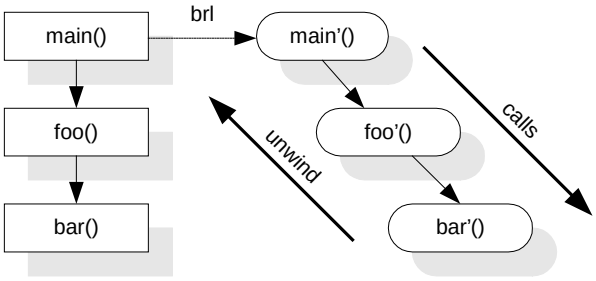


Figure 6: Original and instrumented call graph. Unwinding traverses the call graph upwards.

An example will help to illustrate the problem. Assume function `main()` calls function `foo()`, which calls function `bar()` as shown below:

```
main()          foo()          bar()
{               {              {
  foo();          bar();         . . .
}               }              }
```

After instrumentation, the resulting call stack will contain instrumented functions as indicated in Figure 6.

The IA-64 runtime architecture defines function calls, conventions for passing parameters and preserved registers. To perform a call, a br.call instruction assigns the return address to a branch register –

the return pointer (rp) – before branching to the target address. If the called function (the callee) plans to perform other calls, it has to save and restore the return pointer. On return of a function, a br.ret instruction uses the return pointer to return to the bundle address *after* the bundle containing the original br.call instruction.

The return pointer can be stored in a variety of places: on the memory stack, in a preserved register, in a stacked register, in another branch register. The information where the return pointer is saved is stored in the unwind information present in the process' ELF file and is loaded into the process' address space. The compiler generated code will, of course, reload the return pointer from the correct resource, before performing a function return.

To uninstrument a process at any given time, an IP residing in an instrumented function `bar()'` must be reset to the corresponding IP in the non-instrumented function `bar()`. If `bar'()` has been called from the instrumented function `foo'()`, the return pointer will contain an address within the instrumented `foo'()`. If the return pointer has already been saved as described above, then the resource holding the return pointer will hold an instrumented address.

On returning from `bar()`, no branch back to its instrumented caller `foo'()` must be taken, because the shared memory segment containing `foo'()` may already have been unmapped. To avoid the resulting page fault, the call chain has to be *cleaned up*.

To cleanup the return pointers in the call chain, we have to traverse the current call graph upwards and patch the storage locations for return pointers holding instrumented addresses with their corresponding non-instrumented addresses. This ensures that the code restoring the return pointer before the function return will assign a non-instrumented address to the return pointer before executing the br.ret instruction.

Traversing the call chain upwards is called "stack unwinding". Supporting stack unwinding through instrumented code space is a non-trivial task and will be described in the next sections. Once such support is in place, an even wider class of applications can be supported, including C++ applications that throw C++ exceptions and applications that perform explicit stack unwinding. Because of its close relation to stack unwinding, the support for C++ exceptions will be addressed in parallel [12][17][22].

5.1 Stack Unwinding Through Instrumented Functions

On HP-UX, the functionality to support stack unwinding is located in the system library libunwind. To unwind the call stack on the IA-64, both the memory stack and the register stack have to be unwound in a consistent manner. The IA-64 runtime architecture uses the *unwind information* for this purpose.

It is the responsibility of the compiler to generate unwind information for functions. The unwind tables contain the necessary information to unwind the memory stack and to restore values of preserved registers. For example, the unwind descriptors describe when and where a preserved register is saved to another register or when it is spilled to the memory stack. *Preserved* registers must remain unmodified across function calls. For a given load module, unwind information resides in an executable (ELF file) in several sections as shown in Figure 7.

The unwind information may have a Language Specific Data Area (LSDA) and pointers to routines (*personality routines*) to perform language specific tasks. For C++ exceptions, this involves calling the destructors for local objects as the stack is unwound.

The *Unwind Table* has entries for every region of code that handles preserved registers, which include the return pointers. The entries in the table are sorted for fast binary search. Each entry has a pointer into a potentially unsorted array of *Unwind Information Blocks* (UIB). A single UIB contains source language independent unwind descriptors and a language specific data area (LSDA).

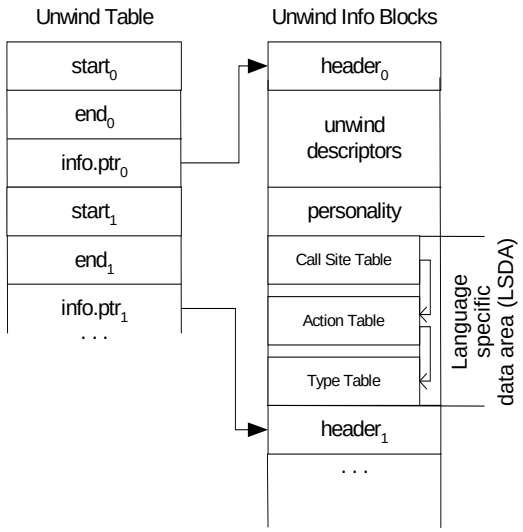


Figure 7: Layout of Unwind Table and Unwind Information Blocks

A large variety of unwind descriptors and formats allows compact storing of the unwind information. Typical cases are encoded in just a few special formats using only a couple of bits. An instruction location is encoded as a slot index *t* relative to the beginning of the the unwind region. This index *t* and other numbers are encoded using a compressed encoding scheme. Example 1 shows the first instructions of a function foo().

```
foo::
0000 alloc     r35=ar.pfs,0,9,2,0
0000 mov       r36=rp
0000 adds      r9=0,sp

0010 addl      r39=0,r1;;
0010 adds      sp=-32,sp
0010 adds      r37=-16,r9
. . .
```

Example 1: An example function entry point in IA-64 assembly language

The corresponding decoded unwind descriptors are shown in Example 2:

```
 1 R1prologue    rlen=6
 2 P7rp_when     t/spoff/pspoff=1
 3 P3rp_gr       gr/br=36
 4 P7pfs_when    t/spoff/pspoff=0
 5 P3pfs_gr      gr/br=35
 . . .
```

Example 2: Decoded unwind information

The system unwind library will interpret these unwind descriptors as:

[1] The code has a prologue region consisting of six (6) instructions.

[2,3] The return pointer is saved to general register `r36` in slot 1.

[4,5] The `ar.pfs` register is saved to general register `r35` in slot 0.

The unwind descriptors that were generated by the compiler for the original function may not match the instrumented function due to the probe code inserted. Therefore the unwind descriptors must either be updated or new unwind descriptors must be generated.

If none of the instructions that were encoded in the original unwind tables are deleted, and only probe code that does not affect stack unwinding of preserved register state is inserted, then a dynamic update of existing unwind information is sufficient. The indices of instructions encoded in the unwind descriptors will have to be updated to reflect their new indices in the instrumented version.

In most cases, the new index values will be higher than the original values due to the additional probe code inserted, and may not fit in the original n bits that were used to represent the old index value. The size of unwind descriptors will therefore have to change during the update. The encoded numbers in the LSDA may also need to grow during updating.

Since the UIBs are physically organized as an array, an UIB entry cannot grow in size in its current location without overwriting subsequent entries. Therefore UIB blocks must be created and placed in new locations. HP Caliper dynamically copies and updates the UIB blocks.

5.2 Pseudo Modules

To start the unwinding process, the unwind library performs a call to dlmodinfo() in the dynamic loader to obtain load module information and the location of the related unwind information. The relocation of instrumented functions to a different address space (shared memory region in the case of HP Caliper) will cause this mechanism to fail as relocated code addresses will not be found in the registered set of load module addresses in the dynamic loader.

We therefore augmented the dynamic loader to allow registration of *pseudo modules*. These pseudo modules are used to register an instrumented function along with its unwind information to the dynamic loader. This registration in one place allows easy and effective synchronization and eliminates the need to update the unwind tables (This mechanism will only handle shared bound executables. Shared binding is the default on HP-UX, and we won't discuss the case of archive bound executables in this paper).

The new dynamic loader interface has to be called from the context of the target application. Since this application is not recompiled or relinked in any special way, there is no direct support for this operation. We therefore add a *prologue* code sequence to instrumented functions. This code will be executed only once per instrumented function and will perform the necessary registration calls to the dynamic loader.

The registration prologue sequence consists of about 80 bundles. Generating a separate prologue for every function will result in a significant code bloat for instrumented functions. Since this code needs to be executed only once per function, we decided to reuse the allocated space for a single prologue sequence between all instrumented function registrations. It must be noted that although we use the same space to store the prologue code, the prologue instruction sequence will be different for each function to be registered.

5.3 Observations

This approach resulted in a couple of interesting problems:

Threads Consider the following scenario for multi-threaded programs running under HP Caliper. Thread A is executing the prologue for a reached and instrumented function foo'(). Another thread B reaches an uninstrumented function bar(). HP Caliper instruments bar(), and then proceeds to overwrite the currently active prologue region. This leads to a race condition. Therefore, each thread allocates its own prologue code space, thereby eliminating the above problem.

Recursive invocation The registration functions called directly by the prologue code may call other functions. If one of these functions has not been instrumented yet, the instrumenter would instrument this new function, resulting in a recursive generation of the prologue. A second prologue generation would overwrite the original prologue. This particular condition can be detected and additional prologue code buffers could have been allocated. However, in multi-threaded applications, this recursive prologue execution could result in a dead-lock. Therefore, recursive invocations are suppressed by not generating prologue code for other functions called by the main registration functions.

Context The invocation of the prologue code must not alter the program behavior. Since the prologue is executed at a function call, we presumed that according to the runtime convention we would only have to save and restore the preserved registers in the prologue. However, three problem cases were identified.

- The registers f8-f15 are used for incoming floating point arguments. The additionally inserted calls to the new API modify these registers, which therefore must be saved and restored in the prologue.
- The registers r8-r11 are used to return non floating-point values up to 32 bytes. The compiler, knowing the return types of functions, may assume that some or all of these registers remain unmodified across a function call. However, the newly introduced function calls may very well modify one or more of these registers. Therefore, these registers have to be saved and restored by the prologue.
- Applications that don't follow the runtime conventions are a potential problem. A common trick is to keep a value in a non-preserved register across a function call, "knowing" that the callee will not modifiy the particular register. However, the inserted functions in the prologue may modify these registers, resulting in altered program behavior. There are two possible solutions to this: One is to save and restore the complete context on entry and exit of the prologue, and the other is to only support programs that strictly follow the runtime conventions. Both solutions seemed non-optimal. Fortunately, there is a remedy - lazy unwind generation.

Lazy unwind generation The descriptions up to this point assumed that the unwind information gets registered via the prologue code just before executing the newly instrumented function. However, this does not have to be the case. Instead, we can register updated unwind information to the dynamic loader in a lazy fashion. For example, only after a C++ exception has been thrown, or only after uninstrumentation has been requested, we could insert and execute a modified prologue that registers all functions that have been instrumented up to this point. This approach will avoid the problems discussed earlier.

Invalid Unwind Information A common technique to achieve the best performance for individual functions is to compile the functions with highest optimization levels and to generate an assembly file. The generated assembly file will contain unwind information for the original functions. When these assembly instructions are hand-tweaked to further optimize the code, the corresponding unwind information is usually not manually updated to reflect the code changes. This basically leads to invalid unwind information, which has to be handled gracefully. In order to enable successful stack

unwinding, functions with invalid unwind information must not be instrumented.

6. EXAMPLE

In this section we discuss the use of dynamic binary instrumentation in the context of profile based optimization (PBO) and discuss the instrumentation specific details. We will use this example to measure the runtime overhead for this particular type of instrumentation under HP Caliper.

Profile based optimization plays an important role in performance tuning of applications. The dynamic instrumentation framework in HP Caliper is used to provide an execution profiler that measures the total execution and taken frequencies of all branches. This data is fed back to the optimizer to direct optimizations such as hot/cold procedure splitting and inlining.

In order to make these measurements, counter code has to be inserted before every branch instruction. These counters are allocated in shared memory and each counter is associated with a source and target address corresponding to the instruction address of the branch and its target.

Branches can be classified as ip-relative branches which have a fixed target that is specified in the branch instruction, and indirect branches in which the branch target is specified through a branch register and can only be determined at run-time.

Allocating counters and adding probe code to profile ip-relative branches is straightforward. The counter for an ip-relative branch and its (single) target is allocated when the function is instrumented.

Similarly, if the number of possible targets for an indirect branch is known through the help of compiler annotations, then the corresponding counters for these indirect branches can be allocated statically. Such indirect branches include those that obtain their targets from switch tables located in the binary. In this case, a separate counter is allocated for each target corresponding to a switch table entry. The switch table target address entries are replaced with the addresses of trampoline code that increments the counter corresponding to this target. Such indirect branches can therefore be profiled easily.

Profiling of indirect branches whose targets cannot be known in advance (for example, indirect call branches) pose the following challenges:

Counter lookup - The target of the branch needs to be determined at runtime and the corresponding counter located and incremented. This problem was solved by inserting probe code before the indirect branch to make the call to the counter lookup and increment function.

Since the probe code introduces a new function call, this requires saving of all registers that might potentially be modified across the call. To eliminate the saving and restoring of scratch registers around the inserted function call, the counter lookup function is compiled with a special flag to ensure that it does not use any scratch registers within the counter lookup routine. Only local stacked registers are used within this function. This greatly reduces the number of registers to be saved across the call.

The counter routine needs to be within the target program's address space since the probe code will be executed in the target program's context. We achieve this by copying the compiled routine into the shared memory space.

Counter allocation - Since the number of possible targets of the indirect branch is not known, the counter data structures need to be allocated dynamically. If the counters are allocated statically, it would result in either a wastage of counter space or it would result in some of the targets of the branch not being profiled (if an insufficient number of counters were allocated). We therefore transfer control to HP Caliper to dynamically expand the allocated counter as new targets are encountered during execution.

It should be noted that the profiling of indirect branches is very expensive due to the execution of the counter lookup routine for each indirect branch counter increment. Control transfers to Caliper to dynamically allocate a new set of counters are even more expensive, but don't happen too often.

Minimizing counters - Several algorithms exist [4] to minimize the counter increments needed at runtime to obtain the total and taken execution frequencies of every branch. We place a function call counter at function entry points and a taken counter for every control transfer instruction (chk.s, chk.a, and all branches) and record all basic block boundaries. The total counts for each basic block (and therefore the branches) can be derived from the above measured counts in a post-processing step.

6.1 Performance

Table 1 describes the set of test appplications that we ran under HP Caliper to measure the performance overhead for differents types of instrumentation.

Application	Description
finite	a commercial finite element solver consisting of about 900.000 lines of C, assembler and FORTRAN code
mgrid	a SPEC2000, FORTRAN 77 application
facerec	a SPEC2000, FORTRAN 90 application
twolf	a SPEC2000, C appplication
eon	a SPEC2000, C++ application

Table 1: The test applications

Table 2 contains the results. We analyzed these key numbers:

- Ratio of functions found and reached (*R%*)
- Total run-time overhead - This includes the overhead caused by the execution of the additional probe code inserted and the overhead associated with the instrumentation process itself (*Over*).
- Run-time overhead caused by the inserted probe code. We were able to derive this number precisely by using the PMU on the IA-64. We measured the CPU cycles spent in the application, when run native and under instrumentation (*OverC, OverF*).
- Total nop instructions in the instrumented code vs the total nop instructions in the original code. This gives some hints on the effects of our simplistic scheduler (*OverN*).

As can be seen from Table 2, the overhead associated with profiling all branches in the program can be significant for short running programs. It is more acceptable for longer running programs like finite, where the overhead associated with executing the inserted probe code will dominate. Generally, we see the factors for the PBO instrumentation ranging from 1.2x to 3.3x.

App	Detect.	Reach	R %	Native	Instr.	Over	C_n	C_i	OverC	*OverF*	N_n	N_i	OverN
finite	4417	461	11	3705	4441	1.2x	*2.9e12*	*3.3e12*	**1.1x**	*1.0x*	*70e9*	*1.3e12*	1850%
mgrid	667	157	26	209	337	1.6x	*166e9*	*266e9*	**1.6x**	*1.0x*	*122e9*	*165e9*	35%
facerec	800	220	26	339	677	2.0x	*268e09*	*520e09*	**1.9x**	*1.1x*	*140e9*	*214e9*	52%
twolf	276	83	30	26	66	2.6x	*19e09*	*49e09*	**2.6x**	*1.1x*	*5e9*	*18e9*	360%
eon	2062	654	31	117	396	3.3x	*92e09*	*243E09*	**2.6x**	*1.3x*	*39e9*	*81e9*	207%

Table 2: Performance numbers

Detected - Number of detected functions
Reached - Numner of reached
R % - Percentage of reached functions
Native - Native runtime in [sec]
Instr - Runtime with instrumentation in [sec]
Over - Overhead in Factors x for branch profiliing

Cn - Native CPU_CYCLES
Ci - CPU_CYCLES with probe code
OverC - Overhead caused by PBO probe code
OverF - Overhead caused by Function Counting probe code
Nn - Nops executed natively
Ni - Nops exexcuted under instrumentation
OverN - Percentage of additionally executed nops

It can be seen that the number of executed nop instructions in the instrumented code is much higher than in the original code. There are several optimizations that can be done:

- Preserve the original instruction bundling (when possible), by placing the inserted probe code in entire bundles.
- Remove all nops in the original code.
- Improve our scheduler.

The above enhancements are expected to improve the compactness of the code, and also improve the runtime of the instrumented code.

Please note that the overhead of executing the instrumented program will vary depending upon the type of measurement to be made. For example, if the measurement being done was function counting (rather then branch profiling), the overhead would be significantly lower (as can be seen from the column *OverF* in Table2).

7. RELATED WORK

A lot of work has been done in the areas of both static and dynamic instrumentation, as well as in simulation of binary executables. Comprehensive collections of approaches, tools and techniques can be found in [8], and more related work is given in [20],[21] and [35].

The two other major projects involved in dynamic instrumentation are the University of Wisconsin's Paradyn group and IBM's Dynamic Probe Code Library (DPCL) project.

Paradyn is a performance measurement tool for parallel and distributed programs. It includes an abstract, machine independent, dynamic instrumentation API (DynInst), and provides precise performance data down to the procedure level. Paradyn has been ported to multiple platforms and has recently released papers on kernel and Java instrumentation [7][15][24][25][31][33].

IBM's DPCL project [13] utilizes work from the Paradyn project and standardizes a C++ interface class library to foster generation of performance tools using dynamic instrumentation.

No related work can be found on binary instrumentation on the IA-64 - it is the purpose of this paper to fill this gap.

8. ACKNOWLEDGEMENTS

We would like to thank our team members and colleagues for fruitful and creative discussions in the cubicle maze. We thank Tara Krishnaswamy for the encouragement to write this paper. Additionally we thank all those engineers at HP that helped with their quality work to make HP Caliper a success. We also thank our anonymous reviewers for their invaluable feedback.

References

[1] A.Aho, R.Sethi, and J.Ullman, "Compilers: Principles, Techniques, and Tools", Mass.: Addison-Wesley, 1985.
[2] J.Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint-Malo, France (October 1997): 1–14.
[3] V.Bala, E.Duesterwald, and S.Banerji, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo" HP Labs Technical Reports, HPL-1999-78, 990621, 1999
[4] T.Ball, J.R.Larus, "Efficient Path Profiling", Processdings of MICRO 96, pages 46-57, December 1996
[5] E.B.Betts, D.P.Hunter, and S.L.Smith, "Moving Atom to Windows NT for Alpha," Digital Equipment, Jan. 1999.
[6] Chrstina Cifuentes, Antoine Fraboulet "Intraprocedural Slicing of Binary Executables", University of Queensland, Australia.
[7] Chrstina Cifuentes, Mike van Emmerik "UQBT – A Resourceable and Retargetable Binary Translator", University of Queensland, Australia (December 1999)
[8] Robert F.Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", Sun Micosystems Laboratories, Incorporated, and the University of Washington, technical report SMLI 93-12 and UWCSE 93-06-06, 1993
[9] R.Cohn, D.Goodwin, P.G.Lowney, "Optimizing Alpha Executables on Windows NT with Spike", Digital Technical Journal 9, 4
[10] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," The USENIX Windows NT Workshop Proceedings, Seattle, Wash. (Aug 1997): 17–24
[11] S.LGraham, P.B.Kessler, M.K.McKusick "An execution profiler for modular programs", Software-Practice and Experience, 13:671-685, 1983
[12] Christoph de Dinechin "C++ Exception Handling for IA-64". USENIX, First Workshop on Industrial Experience with Systems Software., October 22nd, 2000
[13] Dynamic Probe Class Library, The Parallel Tools Consortium, http://www.ptools.org/ (current Sept 2001).
[14] Susan L.Graham, Steven Lucco, Robert Wahbe, "Adaptable Binary Programs", Usenix 1995
[15] Jeffrey K.Hollingsworth, Barton P.Miller, "Dynamic Instrumentation API", Journal, University of Wisconsin, 1996.
[16] Intel Corperation "IA-64 Application Developer's Architecture Guide", May 1999
[17] ISO C++ Final Draft International Standard (Now ISO standard)

[18] Hewlett-Packard Company, “IA-64 Software Conventions and Runtime Architecture”, Version 1.0, August 31, 1999
[19] Hewlett-Packard Company, “HP Caliper”, http://www.hp.com/go/hpcaliper, (current Oct, 8th, 2001)
[20] R.Hundt “HP Caliper – An Architecture for Performance Analysis Tools”. USENIX, First Workshop on Industrial Experience with Systems Software., October 22nd, 2000
[21] R.Hundt “HP Caliper – A Framework for Performance Analysis Tools”. IEEE Concurrency, Vol 8, Num. 4, Oct-Dec 2000, Los Alamitos, CA, 90720
[22] R.Hundt, V.Ramasamy "Dynamic Instrumentation of C++ Applications on IA-64", International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2001, June 25 - 28, 2001, Las Vegas, Nevada
[23] J.R.Larus and E.Schnarr “EEL: Machine Independent Executable Editing. In SIGPLAN Conference on Programming Languages, Design and Implementation, pages 291-300, June 1995
[24] Barton P.Miller, Mark D.Callaghan, Jonathan M.Cargille, Jeffrey K.Hollingsworth, R.Bruce Irvin, Karen L.Karavanic, Krishna Kunchithapadam and Tia Newhall “The Paradyn Parallel Performance Measurement Tools” IEEE Computer 28, 11, pp.37-46 (November 1995).
[25] Tia Newhall and Barton P.Miller “Performance Measurement of Interpreted Programs” Europar'98, Southampton, England, September 1998.
[26] Rational Software Cooperation. Product documentation for Purify, Quantify and PureCoverage.
[27] Ted Romer et al. “Instrumentation and Optimization of Win32/Intel Executables using Etch”, Usenix Windows NT Workshop 1997
[28] M.Rosenblum et al., “Complete Computer System Simulation: The SimOS Approach” IEEE Parallel and Distributed Technology, Vol.3 No.4, Winter 1995
[29] A.Srivastava and A.Eustace, "ATOM: A System for Building Customized Program Analysis Tools," Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation, Orlando, Fla. (June 1994): 196–205.
[30] Standard Performance Evaluation Cooperation (SPEC), “Spec CPU2000”, www.spec.org (current Nov. 2000)
[31] Ariel Tamches and Barton P.Miller “Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels” Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, February 1999.
[32] M.Weiser “Program Slicing”. IEEE Transactions on Software Engineering, SE-10(4):352-257, July 1984
[33] Zhichen Xu, Barton P.Miller and Oscar Naim “Dynamic Instrumentation of Threaded Applications” 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Atlanta, Georgia, May 1999
[34] Cindy Zheng, Carol Thompson “PA-RISC to IA-64: Transparent Execution, No Recompilation”, IEEE Computer Society Cover Feature, 3/2000
[35] Richard A. Uhlig, Trevor N. Mudge “Trace-Driven Memory Simulation: A Survey”, ACM Computing Surveys, Volume 29, Issue 2, (1997)

Tritanium: Augmenting the Trimaran Compiler Infrastructure To Support IA-64 Code Generation

Yogesh Chobe, Bhagi Narahari, Rahul Simha
Department of Computer Science
The George Washington University
Washington DC 20052
{ylchobe, narahari, simha}@seas.gwu.edu

Weng-Fai Wong
Department of Computer Science
National University of Singapore
3, Science Drive 2
Singapore 117543
wongwf@comp.nus.edu.sg

ABSTRACT

This paper describes Tritanium, an on-going project to enhance compiler optimization tools for Explicitly Parallel Instruction Computing (EPIC) architectures, in particular, the Intel's IA-64 Itanium processor [1]. The project we describe involves augmenting a popular research compiler suite, the Trimaran compiler infrastructure [3], with the capability to generate IA-64 code so that its well-documented capabilities can be used to explore compiler optimizations by testing the resulting code directly on the IA-64 processor. We have validated the tool with preliminary SPEC-based results that show performance comparable with gcc and sgicc; this suggests that the tool can be used for exploring and testing compiler optimizations to improve performance.

1. INTRODUCTION

In this paper, we describe the Tritanium (Trimaran for Itanium) project: an effort to enhance the Trimaran research compiler to produce code for the IA-64 EPIC processor. Since EPIC processors rely strongly on the compiler to extract instruction-level parallelism [8], an infrastructure that provides the ability to experiment with compiler optimizations helps compiler-writers save time in algorithm development. The Trimaran compiler suite is one such popular research compiler that allows experimentation at three levels: front-end compiler techniques (for example, loop unrolling, program transformations, etc), back-end resource optimizations (for example, scheduling and register allocation), and architectural exploration (for example, changing the number of registers, etc) [3]. Unfortunately, the architecture for which Trimaran was developed (a precursor to IA-64) was not realized in hardware and remains inside Trimaran's simulator. Therefore, it is difficult to obtain realistic measurements. We believe the compiler community, in particular, the growing Trimaran community, would benefit from the ability to have the tools generate code directly for the IA-64.

The importance of tools such as Trimaran and Tritanium is well-established. EPIC architectures, including the IA-64, provide hardware features that have the potential to exploit high levels of Instruction Level Parallelism (ILP). Because EPIC processors rely heavily on the compiler to extract parallelism and perform instruction scheduling, the compiler is now the focus of experimentation in optimization strategies. Since experimentation requires easy access to the internals of the compiler and the ability to measure performance at various stages of the compiler process, Trimaran also consists of a full suite of analysis and optimization modules, as well as a graph-based intermediate language. Optimizations and analysis modules can be easily added, changed or bypassed, thus facilitating highly targeted compiler optimization research.

This paper describes the design issues involved in extending and modifying the Trimaran Infrastructure to generate code for the Itanium processor. These include using the hardware-description langugage of Trimaran (MDES) to support Itanium, providing a code-generator and handling issues specific to IA-64 in the compiler front-end. The performance of the Tritanium is being evaluated by comparing its performance to that provided by gcc and sgicc on a number of benchmarks. Current results include some of the SPEC benchmarks with work going on to make the compiler robust enough for more. These results indicate that the Tritanium infrastructure produces code whose performance is comparable to that produced by gcc and sgicc.

2. TRIMARAN - A BRIEF DESCRIPTION

Trimaran consists of the following components.

- An architecture description language called MDES [5][6] to describe the target processor.
- A compiler front-end for the C programming language, called IMPACT, which performs parsing, type checking and a large suite of high-level (i.e. machine independent) classical and ILP optimizations.
- A compiler back-end, called Elcor, parameterized by a machine description, that performs instruction scheduling, register allocation, and machine-dependent optimizations.

- An extensible IR (intermediate program representation) that has both an internal and textual representation (called Rebel), with conversion routines between the two. This IR supports modern compiler techniques by representing control flow, data and control dependence, and many other attributes.
- A cycle-level simulator of the HPL-PD architecture, which is configurable by a machine description and provides run-time information on execution time, branch frequencies, and resource utilization. This information can be used for profile-driven optimizations as well as to provide validation of new optimizations.
- An integrated Graphical User Interface (GUI) for configuring and running the Trimaran system. Included in the GUI are tools for the graphical visualization of the program intermediate representation and of the performance results.

Trimaran allows various optimization modules to be inserted into the optimization path; thus providing researchers with an infrastructure with which to test compiler optimization algorithms on the HPL-PD architecture. For example, researchers can test a new register allocation algorithm by replacing the built-in register allocation module with their algorithm. The proposed Tritanium infrastructure, as a result of it being developed on top of the Trimaran infrastructure, provides the same flexibility and thus allows researchers to test their optimization algorithms on the Itanium/IA-64 architecture.

3. IA-64 OVERIEW

The Intel Itanium is an IA-64 EPIC architecture with a 64 bit processor supporting most of the EPIC features provided in HPL-PD [1] [7]. However, there are some major differences between the two architectures. For example, Itanium does not provide a division instruction and has a different register stack engine. Besides this, other differences include the 32 bit addressing of HPLPD and the 64 bit addressing of the IA-64 processor. In summary, the IA-64 supports features such as two system environments (32 and 64 bit modes), hardware support for speculation (control and data), predication, and rotating registers. In addition to predication, the compiler can use branch predict instructions which can communicate early indication of target address. IA-64 avoids the unnecessary spilling and filling of registers at procedure call and return interfaces through compiler controlled register renaming. Compiler optimizations which utilize these EPIC features can be developed and tested on Trimaran for the HPL-PD and on the Itanium processor using our proposed augmentation- Tritanium.

4. DESIGN AND IMPLEMENTATION OF TRITANIUM- AN OVERVIEW

Tritanium has been implemented by writing a back end code generator module that accepts Elcor output and writes out the IA-64 assembly file. The tasks involved in the implementation of Tritanium also required changes to other existing modules in Trimaran as listed below:

- Creating a hardware description of IA-64 in MDES. The Trimaran Infrastructure uses MDES, a high-level machine description language, to describe the target processor. The IA-64 had to be described as a first step towards retargeting the compiler for IA-64. The number of registers, number of functional units and their latencies had to be described. Operations that had characteristics different from HPL PD were described in the modified Elcor file along with some changes to follow the runtime conventions of IA-64.
- Changes to IMPACT (the front-end) to support the runtime conventions of IA-64 and branch semantics. Impact is the front end compiler in Trimaran that performs machine independent optimizations prior to resource allocation. Changes were made to the instruction selection module to generate code that can be mapped to the IA-64 processor. The runtime conventions of the IA-64 for parameter passing also had to be built into this stage.
- Supporting the rotating registers of the IA-64 in Elcor (the back-end). HPL-PD assumes a separate set of rotating registers from regular registers. This is not true for the IA-64. Hence in the post-processing phase of modulo scheduling, the current predicate registers are saved before rotating. In Elcor, two registers are considered equal even if one is rotating and other is not. However, this is not the case in IA-64 and therefore affects the correctness of dependency inferences and the correctness of scheduling. Hence changes were made to distinguish between them.
- Supporting instructions that don't have hardware implementation on the IA-64. We included code to replace instructions such as DIV and MOD, which do not have hardware implemention on the IA-64 by generating the appropriate routines to perform such operations at the instruction selection phase in IMPACT.
- The IA-64 code generator module. The output generated from Elcor is similar to IA-64 code, but is still in the syntactical representation of HPL-PD. This output was transformed to the IA-64 assembly file by the Code generator module. Essentially, the code generator module maps the HPL-PD instructions to the equivalent IA-64 instructions.

5. VALIDATION OF TRITANIUM

The Tritanium system was validated initially using the NUE [2] (Native User Environment) Simulator from HP and later using an Itanium SDV machine provided by Intel Corporation. The following Benchmarks / Test Suites were used while developing Tritanium to test the implementation validity of the compiler.

1. Trimaran Test Suite: The test programs that came with Trimaran were the initial targets for the compiler. This suite includes small programs that test different aspects of the compiler and various aspects of the language.
2. SPEC Benchmarks: Spec (The Standard Performance Evaluation Corporation) is the publisher of benchmark suites that aim to provide a standardized set of relevant benchmarks that can be applied to obtain a performance evaluation of architectures.

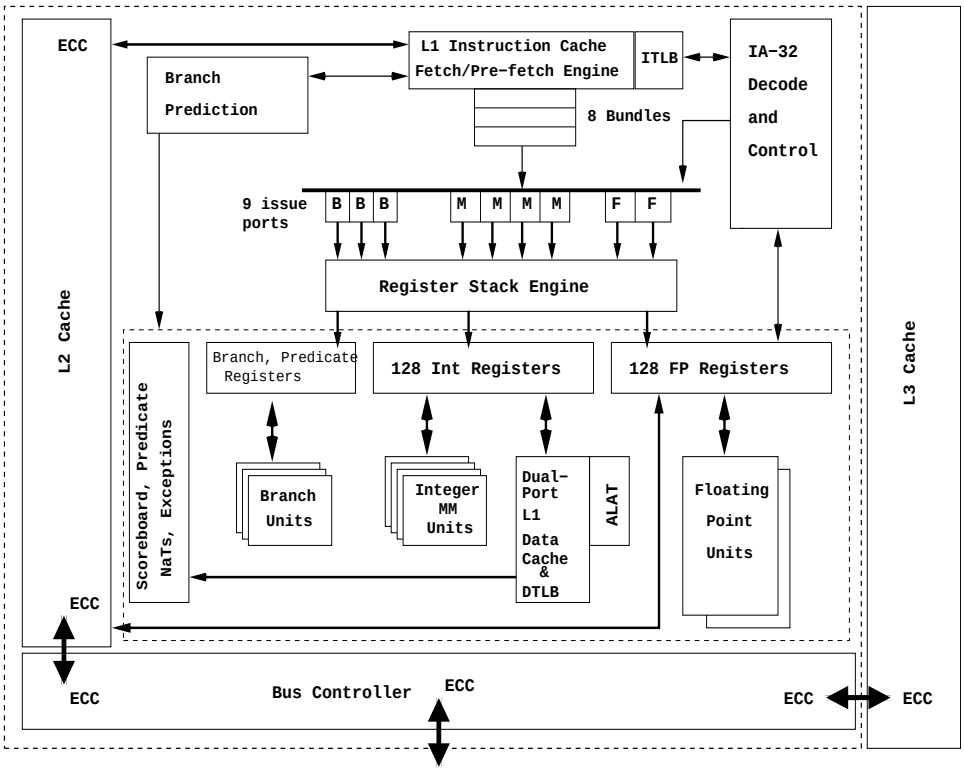


Figure 1: Block Diagram of Itanium Processor

3. MediaBench: MediaBench is a suite of benchmarks from UCLA, which is provided as a tool for evaluating and synthesizing multimedia, and communication systems.

4. DIS: The Data Intensive Systems Benchmark Suite developed by the Atlantic Aerospace Electronics Corporation and the Boeing company and ERIM Int. Inc.

5. Olden: The benchmark suite from Princeton which was used to test Olden, which is the language and runtime system for parallelizing programs using dynamic data structures.

Currently, we have not completed testing of all the programs in the benchmarks suites but anticipate completion incrementally as the infrastructure is released. The list of currently validated benchmarks will be available at the Tritanium website [4]. The performance of the code generated by the Tritanium infrastructure (using the optimization modules currently available in Trimaran) for some test programs is presented in the table below. As the table shows, the performance of Tritanium is comparable to gcc with the current implementation, which has not yet been tuned for performance. With further work on tuning, which will include some basic required additions like support for superblock and hyper block region formation, due to which we expect an improvement in performance.

For the performance table above, we compiled the programs using gcc with the -O3 option and sgicc with the -O3 option. These options correspond to the full range of optimizations

Table 1: Performance Comparison of Tritanium

Test Program	Tritanium	gcc	sgicc
052.alvinn	115	117	100
022.li	102	100	101
026.compress	112	110	100
124.m88ksim	108	116	100

provided by these compilers - for example, sgicc in the -O3 option forms hyperblock regions. In constrast, the Tritanium compiler was used with the basic block region formation option. At the time of running the tests, the superblock and hyperblock region formation techniques were not stable enough (for IA-64 code) to be included in the tests. The numbers in the table are based on the execution times on the Intel Itanium SDV. The entries in the table show the relative performance of the compilers, on each benchmark application, and not the absolute execution time of the programs. The numbers in the table were obtained by dividing all the execution times with the least execution time of that program - the lowest execution time was almost always obtained with sgicc and thus the sgicc performance has been normalized to 100%.

6. CONCLUSIONS AND FUTURE WORK

The Tritanium infrastructure will allow compiler researchers to develop, test, and validate ILP optimization techniques by examining performance of their techniques on the Itanium processor. Currently, we have been able to compile a number of small programs and some applications from var-

ious benchmark suites using the base set of optimizations provided in Trimaran. The generated code, inspite of using some conservative optimizations, shows performance comparable to other compilers like gcc. We expect to support the full range of region formation methods, such as superblock and hyperblock, soon. In addition, enhancements to Tritanium will include some basic required additions such as support for compiling varargs functions, and additions of some unimplemented instructions. While the initial phase of Tritanium development involved generating correct code and making the compiler compliant with the standard benchmarks, the next phase involves performance tuning of the compiler.

7. ACKNOWLEDGMENTS

The authors would like to thank Intel Corp for the Itanium SDV machine given to The George Washington University. The authors would also like to thank Aakash Kambuj, formarly at National University of Singapore for his contributions in the initial implementation of Tritanium.

8. REFERENCES

[1] The itanium architecture manuals. *http://developer.intel.com/design/itanium/manuals/index.htm.*

[2] Nue, ia-64 simulator. *www.software.hp.com/ia64linux.*

[3] Trimaran web site. *www.trimaran.org.*

[4] Tritanium web site. *www.seas.gwu.edu/ťri and www.seas.gwu.edu/ÿlchobe/tritanium.html.*

[5] S. Aditya, V. Kathail, and B. R. Rau. Elcor's machine description system: Version3.0. *HPL Technical Report HPL-98-128. Hewlett-Packard Laboratories*, July 1998.

[6] J. C. Gyllenhaal, W. m. W. Hwu, and B. R. Rau. Hmdes version 2.0 specification. *Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign*, 1996.

[7] V. Kathail, M. Schlansker, and B. R. Rau. Hpl-pd architecture specification:version 1.1. *HPL Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories*, February 1994.

[8] M. Schlansker and B. R. Rau. Epic: An architecture for instruction-level parallel processors. *HPL Technical Report HPL-1999-111. Hewlett-Packard Laboratories*, February 2000.